

1/ In a second level of sophistication, voice response systems have additional capabilities, such as "voice forms" capability (where, e.g., a vendor can collect the caller's name, address, and the product the caller wants to order, by recording the caller's voice responses to recorded prompts). These more sophisticated systems also provide the user with the option of transferring to a human operator for further assistance, either by staying on the line for a specified time interval past the end of the computer response (the primary purpose of this feature is assist rotary-dial telephone users or those who become confused and do not enter any tones for the specified interval), at designated times as a menu option (e.g., after a computer response to a query, the user could be given the choice of entering another query or transferring to an operator or another extension, as well as terminating the call), or at any time by pressing a particular key, such as the zero key. The extension to which the call is transferred could be another voice response feature. However, any information the caller gave to the first feature would not be available to the second feature and would have to be given again by the caller.

Because both voice messaging and voice response systems have as options the ability to transfer to other extensions, in an installation having both types of systems it could be said that the systems have been integrated. A caller could always transfer from one system to the other by dialing the other system's extension from the first system, assuming those extensions were known to the caller or included in the system prompts. Thus, a caller attempting to reach a particular individual could leave a message in that individual's mailbox and then transfer to a voice forms feature to order particular merchandise. Conceivably, the caller could then transfer to some other voice response feature. However, each time the caller would have to repeat information previously given. For example, even though the caller gave his or her name in the voice mail message, he or she would have to repeat it for the voice form. At most, there are previously known systems in which, when a caller transfers to a human operator, the operator is informed of where in the voice response system the caller is exiting from (e.g., session screen identification number), without any additional information.

Similarly, a caller may access a voice response feature in which digital information is called for, and then transfer to another feature also requiring digital entries, and be required to enter the same information again.

It would be desirable to be able to provide an integrated voice messaging/interactive voice response system in which information given by a caller could be transferred among various application modules of the integrated system.

SUMMARY OF THE INVENTION

It is an object of this invention to provide an integrated voice messaging/interactive voice response system in which information given by a caller could be transferred among various application modules of the integrated system.

In accordance with this invention, there is provided an interactive voice response system including interface means for connecting the interactive voice response system to one or more incoming telephone lines through which a caller communicates with the system, one or more voice response application modules, each performing a respective voice response feature, and a voice response processor linking the interface means to the voice response application

modules, wherein at least one of the voice response application modules is capable of gathering data from the caller, the caller may transfer among the voice response application modules, and the voice response processor passes at least a portion of the gathered data from the at least one of the voice response application modules to other of the voice response application modules when the caller transfers among the voice response application modules.

DRWDESC:
BRIEF DESCRIPTION OF THE DRAWINGS

The above and other objects and advantages of the invention will be apparent upon consideration of the following detailed description, taken in conjunction with the accompanying drawings, in which like reference characters refer to like parts throughout, and in which:

FIG. 1 is a conceptual representation of an integrated voice messaging/voice response system according to the present invention;

FIG. 2 is a diagram of a preferred hardware configuration of a system according to the present invention;

FIG. 3 is a graphical representation comparing the information content of a telephone call session on a system according to the invention, as compared to a previously known system;

FIG. 4 is a diagram of a preferred hardware configuration of a system according to the present invention including apparatus for programming applications;

FIG. 5 is a diagram of an X.25 packet format used in implementing the system according to the invention; and

FIGS. 6A and 6B (hereinafter collectively referred to as FIG. 6 are a flowchart showing the implementation of the system according to the invention.

DETDESC:

DETAILED DESCRIPTION OF THE INVENTION

In accordance with the present invention, a voice processing system supporting a number of voice messaging and interactive voice response features is linked to a user site Private Branch Exchange (PBX) or central office-based telephone system (such as that known by the name CENTREX). Included voice messaging features might be voice mail and automated attendant, while included interactive voice response features might be "traditional" interactive voice response (query database and get response), voice forms, and voice bulletin boards. Other applications may also be available.

A caller may enter the voice processing system of the invention either through a voice messaging feature after leaving a message for an absent party and then transferring to some other feature, or by dialing directly into a particular feature, assuming that that feature has a direct telephone line or is available through a main menu. Each feature is a separate application module of the voice processing system, and is programmed in accordance with the needs of the particular installation.

The voice processing system unit may be further connected to one or more host computers, as necessary to service the various application modules. The link between the voice processing unit and the host computer is a communications link preferably governed by a conventional terminal emulation protocol such as the IBM "3X74" emulation protocol, although other protocols may be used.

The conceptual layout of system 10 according to the invention is shown in FIG. 1. A caller at originating telephone 11 is connected over the public switched network 12 to the system owner's PBX or the central office serving the system (if the system owner's installation uses a phone company system such as CENTREX) at 13. PBX or central office 13 is connected to the voice processing system unit 14 by link 15 of the type conventionally used for such connections. PBX or central office integration on link 15 may be provided, which allows the system to know the identity of the party called, the reason for forwarding the call to the system (e.g., the caller's original destination is busy), etc., which may be useful for certain voice messaging or interactive voice response features.

Voice processing system unit 14 supports a plurality of different voice processing application modules as determined by the system owner. In the exemplary preferred embodiment shown in FIG. 1, voice processing system unit 14 supports interactive voice response module 140, voice forms module 141, voice bulletin board module 142, automated attendant module 143, voice mail module 144, telephone answering module 145 and message networking module 146. Modules 140-142 are, broadly, interactive voice response modules, while modules 143-146 are voice messaging modules. While the application modules used in voice processing system unit 14 may involve additional hardware, they are preferably implemented within system unit 14 as software.

Voice processing system unit 14 also may be connected to one or more host computers 16, as required by the supported interactive voice response application modules. The connection between system unit 14 and host computer 16 is a link 17 of the type conventionally used for such connections. Communications over link 17 may be carried out under the emulation protocol appropriate to the computer 16, or by some other data communication method such as the protocols described in CCITT Recommendations X.25, 802.3, etc.

A preferred hardware configuration for a system according to the invention is shown in FIG. 2. As seen in FIG. 2, the system according to the invention includes a voice processing system processor unit 20 linked to a PBX or telephone company central office 13 by conventional links 15. A voice processing application processor unit 21, which could be a conventional IBM Registered TM -compatible microcomputer having an Intel Registered TM 80386 or better microprocessor, is linked to processor unit 20 by one or more links 22 (e.g., under the protocol described in CCITT Recommendation X.25). Application processor 21 controls access to the various voice processing application modules in voice processing system processor unit 20 in accordance with software routines. Processor 20 and application processor 21, as linked, together form system unit 14 of FIG. 1. As shown in FIG. 1, system unit 14 may be linked to host computer 16 by link 17.

In accordance with the X.25 protocol, each incoming call is assigned to an X.25 virtual circuit by assigning to it an X.25 "logical channel identifier." As the caller enters various keystrokes, commands are sent back and forth over link 22 between processor 20 and application processor 21, which may emulate a

terminal of host computer 16, allowing application processor 21 to extract and enter data from and into host computer 16. If, while the caller is in a particular application module, he or she enters or speaks certain information (e.g., name, account number), that information, or a memory address indicating where that information is stored, is passed between processor 20 and application processor 21, along with the logical channel identifier, so that the system always knows which call is being serviced and where to find information already given by the caller. If the information given is digital information, it is preferably stored in application processor 21, while voice information is preferably stored in processor 20. However, in systems in which there is more than one application processor 21, all information may be stored in processor 20, so that it is accessible to all modules on all processors 21, in which case processor 20 would pass to processors 21 the storage locations of the data rather than the data themselves.

Thus, when the caller transfers into an application module requiring keystroke data entries, the system checks to see whether or not there are any keystroke data stored as a result of the caller having been in a different application module requiring such entries. Similarly, if the caller has been sufficiently identified, the system may have retrieved and stored data about the caller previously known to host computer 16. If there are such data stored, the system passes appropriate data to the new application module, and the new module does not prompt the caller to enter that data again.

For example, if the system is operating in a mail-order environment, there may be an application module in which a caller can check the status of an existing order, as well as an application module for placing new orders, both of which might require entry of an account or customer number, possibly a password, and possibly other data in common. When the caller enters one of those application modules, the system will check to see if any of the required data for that module, that are of a type that is also required in the other module, have previously been entered. If they have, the system will pass that information to the second module, which then will skip over any prompts it would otherwise give for the caller to enter those data. In this example, if the caller first checked on an existing order, and in so doing entered an account number, a password, and the order number of the order being checked on, the system would store the account number, password and order number in memory as being potentially useful in other modules. The system would also recall from the host computer other information associated with the account number, which would then also be available for other modules. If the caller then transferred to the new order module, the system would automatically enter the account number and password, which otherwise might be the first two prompts, and the first thing the user would be asked for might be the catalog numbers of the products being ordered. The system would work similarly if the new order module were entered first and the order checking module entered second, as well as between other modules requiring similar keystroke data entries.

The system may also operate in this manner in application modules requiring spoken data entries. For example, a caller may first enter an application module requiring him or her to speak his or her name in response to a first prompt, his or her address in response to a second prompt, and then other module-specific information in response to subsequent prompts. If the caller then transfers to a second application module requiring spoken entries, the system would pass the spoken name and address data (or their location) to the second module, and the caller would not be prompted for that information again. Such a feature could

even be useful when the second "module" is the telephone extension 18 of a human being. In that case, when the human being answers the telephone, he or she hears an introduction announcement before the call is connected, stating that there is an incoming call from the caller, whose name might be spoken in the caller's own voice. If the human being is unavailable, and the caller is transferred to a voice mailbox, a preamble is recorded in the mailbox before the caller's message, identifying the caller, perhaps in the caller's own voice. Additional information previously entered by the caller or known to host computer 16 can also be indicated in the preamble as desired. To make this feature particularly useful, the caller preferably is informed, before being connected either to the human being or the voice mailbox, that it is not necessary to further identify him- or herself; otherwise, he or she would defeat the purpose of the feature by speaking his or her name again.

A telephone call linking several application modules on a system according to the invention, with information passed among the modules, can be referred to as a "compound session". The only prerequisite for establishing a compound session is that the first module called be an interactive voice response module as opposed to a voice messaging module, because establishment of a compound session in the preferred embodiment requires the control of application processor 21. However, this can be guaranteed by routing all incoming callers to an interactive voice response module offering a menu of available modules. The passing of information in a compound session can be referred to as an "introduction"-where the information is passed to a human operator-or a "preamble"-where the information is passed to another module in the system.

The progress of a telephone call session on a system according to the invention such as that represented in FIGS. 1 and 2 is shown in FIG. 3, along with the progress of a call session on a previously known type of system. FIG. 3 is in the form of a graph in which the abscissa represents time and the ordinate represents the information content of the session.

In traditional session 30, the caller has called a traditional audio response unit which might, for example, report the caller's account balance at 3 in response to the keystroke entry of an account number. If the caller requires further information at that point, the system provides the option of transferring to a human operator at 32. After the caller has been assisted by the human operator, the total information value of the session at 33 is higher than it is at 32. However, in going from point 31 to point 33, the session goes through a temporary drop in information content at 34, because there was no way for the system to pass to the human operator the information that the caller has previously entered or information about the caller previously known by the host computer.

On the other hand, for example, in a system according to the present invention, as described below, the human operator, before being connected to the caller at 32, might have heard an introduction including a synthesized spoken version of the caller's name derived from information stored in connection with the account.

In session 35 using the system according to the invention, the information value continually increases as the caller moves from application module to application module. Continuing the mail order example used above, session 35 may start in interactive voice response application module 350 in which the caller can obtain charge account balance information.

Moving to audiotex module 351, the caller could obtain instructions on how to order merchandise, which could be accomplished in interactive voice response module 352. The caller's charge account number, already entered in module 350, is effectively passed (in fact, the data are retained for use by subsequent modules, and the memory locations are passed) to module 352, along with the caller's name and address which are derived from the account number as stored in host computer 16. Thus in ordering module 352 the caller is not prompted for name, address or account number, although he or she would be prompted for such data if he or she calls first to ordering module 352 or not as part of a compound session.

Next the caller can transfer to voice forms module 353 where he or she can request mailing of the newest catalog. Here, the system plays back a computer-generated or -spoken preamble 354 which supplies as much of the information that would be required to fill in the voice form as can be derived from information entered by the caller earlier in the session (such as the caller's account number), or from information known to host computer 16 (such as the caller's name and address). Thus, here the caller is again spared from having to repeat his or her name and mailing address, and need only specify, in response to a prompt, the catalog desired.

After requesting the catalog, the caller transfers to interactive voice response module 355 where he or she can determine the status of a previous order by entering the order number. Whereas a caller calling directly to module 355 might have to enter identifying information to verify that he or she is entitled to know the status of the order, all of the necessary identifying information is passed to module 355 by the system if it has previously been entered.

Finally, the caller may attempt at 356 to transfer to his or her customer service representative. If the customer service representative is available and answers the telephone, an introduction 357 will be played identifying the caller by name. Introduction 357 may also report to the customer service representative the various application modules that the caller has already "visited", to enable the representative to give the caller better service. Further, because of PBX/central office integration at 15, the system can also tell the customer service representative which application module-or person's extension-the caller's call was originally directed to. The caller and the customer service representative then conduct a conversation 358, which ends at the information content level represented by point 359, well above the information content level at point 33.

If at point 356 the customer service representative is not available, the caller is transferred to the customer service representative's voice mailbox. However, before the caller is connected to the mailbox, a preamble 360, similar in content to introduction 357, is recorded into the mailbox. The caller, who is preferably informed that it is not necessary to further identify oneself, can then leave a message at 361. The customer service representative is notified of the message at 362 and plays it back at 364, and the transaction ends at 365, at the same information content level as point 359. Alternatively, the representative on being notified of the message at 362 may determine that a different representative is better qualified to handle the message, in which case the first representative, using message notification or message networking, would send the message to the second representative who would play it back at 365.

9 In the particular session 35 as shown, the caller identifies him- or herself through keystroke entries before any spoken identification is required. Therefore, in all spoken identifications of the caller during the session-i.e., in voice forms module preamble 354, transfer introduction 357 and voice mail preamble 360-a synthesized spoken version of the caller's name would have to be used, assuming the system has voice synthesis capability. It is equally possible according to the invention, however, that the caller will call first to, for example, voice forms module 353, in which case a spoken name in the caller's own voice is available to be used where appropriate.

It should be understood that the order of application modules visited in session 35 is illustrative only, and that users of systems according to the present invention may navigate among the available application modules in any desired order. It should be further understood that the present invention can be used in any type of integrated voice messaging/interactive voice response system, and not only the mail order system described in connection with FIG. 3, which is illustrative only.

As indicated above, in the preferred embodiment the various voice response application modules are run as software routines in IBM Registered TM -compatible application processor 21, which operates in a terminal emulation mode of host computer 16. This facilitates programming of the voice response application modules, as illustrated in FIG. 4. Because the voice response application modules that interface with host computer 16 are simply entering data into fields on screens for particular types of inquiries, programming computers 40, which are also IBM Registered TM -compatible personal computers, can be used to run through the "script" of each type of inquiry on host computer 16 in the same terminal emulation mode via links 41. Using that script, the prompts necessary for the particular voice response application module can be developed, tested and stored on diskettes 42. Using this off-line programming facility, new routines can then be loaded onto application processor 21 to update the various voice response application modules with minimal downtime or other negative effects on other applications or sessions supported by voice processor 20 and application processor 21. It is understood, of course, that instead of transferring new routines on diskettes 42, programming computers 40 could be directly linked (not shown) to application processor 21.

The system according to the invention has thus far been described as having an application processor 21 as part of the system, linked to an external host computer 16. However, it is equally possible that the application processor and host computer are the same processor 21. This is more and more likely as the processing power of so-called "personal" computers increases. Similarly, both the "host" and "application processor" functions could be carried out by a larger processor 16. In either such case, application processor/host 16 or 21 would communicate with processor 20 over a data communication link such as X.25 link 22, and there would be no need for terminal emulation link 17.

In the preferred embodiment, the system functions by having voice processing processor 20 collect DTMF digits and caller voice responses. The DTMF digits are decoded into pure digital data and transferred over X.25 link 22 to application processor 21, while the voice data are stored in processor 20 and their address(es) sent digitally over link 22. Digital data sent to application processor 21 are maintained throughout a session to be used when needed by the various modules. Voice data stored in processor 20 are maintained for a predetermined duration that is passed to application processor 21 so that

processor 21 knows how long the voice data will remain available.

Messages sent over X.25 link 22 are in conventional X.25 format, which is well known to those of ordinary skill in the art. That format includes an identification of the virtual circuit. The content of the data area of an X.25 message packet according to the system of the invention is shown diagrammatically in FIG. 5. X.25 data 50 as a whole constitute a command of the system according to the invention. The command begins with two bytes 51 containing the command length, which tells the receiving processor (20 or 21) how long a command string to look for. Next in the command string are two bytes 52 containing the command identifier, which tells the receiving processor the command type. Next is a single byte 53 known as the "command modifier", which indicates whether the message is a command, a response to a command, or an error message. Finally, there are a variable number of bytes 54 (hence the length given in bytes 51) containing parameters passed for use in implementing the command. The structure of each parameter is shown at 500. Each parameter begins with two bytes 501 identifying the parameter type. Certain types of parameters have fixed length, and their types are sufficient to indicate the length of the remainder of the parameter data. However, for parameter types which take variable length data, byte 502 is provided to indicate total parameter length. Following bytes 501 (or bytes 502 in variable length cases) are the actual parameter data 503. As stated above, the parameters passed can include digital data entered by a caller as well as digital addresses of voice or digital data entered by a caller and stored in processor 20.

The logical flow of an interactive voice response ("IVR") session according to this invention is shown in diagrammatic flowchart form in FIG. 6. The IVR session begins at 601, when the voice messaging system ("VMS") (processor 20) transfers a caller to an IVR mailbox. At that time, as seen in step 602, the VMS sends a message to external application processor ("EAP") 21 saying that the session has begun. The EAP now has control of the session, and at step 603 the EAP sends a command to the VMS. Alternatively, the EAP could have started the session by telephoning the "caller" through the facilities of the VMS (e.g., in a telemarketing application). Once the "caller" answered, he or she would be connected to the IVR mailbox as at 601.

The VMS begins processing the command at step 604. The first processing step at test 605 is determining whether or not the command is a command to "visit" a mailbox (either to leave a message for a person or to fill in a voice form). If, at test 605, the command is not a visit command, then at test 606 the VMS tests to see if it is a command to transfer the call to a voice extension. If, at test 606, the command is not a transfer command, then the VMS carries out the command at step 607 (e.g., collect digits or play phrases for the EAP application) and sends a response to the EAP at step 608 indicating completion of the command.

At test 609, the system determines (based, e.g., on user inputs or other criteria, as determined by the EAP programming) whether or not the session is to continue. If the session is to continue, the system returns to step 603 and the EAP sends a new command. If, at test 609, the session is not to continue, then at step 610 the VMS makes the appropriate disposition of the call according to the command sent by the EAP in accordance with its programming, and the session ends at 611.

If, at test 605, the command is a visit command, then at step 612 the VMS stores the logical channel identifier, the session identifier, the application type, and a return location, so that at the end of the visit it will know to where to return. The VMS then connects the caller to the mailbox to be visited at step 613, which could be an ordinary mailbox or a voice form or other type of mailbox. At step 614 the caller interacts with the mailbox, entering all the commands normally associated with the mailbox to erase, re-record, playback or, finally, send the message and exit the mailbox. Once the caller indicates that he or she is ready to send the message and leave the mailbox, then at test 615, before actually sending the message and leaving the mailbox, the VMS tests to see if the EAP had passed it any preamble data. If the EAP had passed preamble data to the VMS, at step 616 the VMS prepends that preamble data to the message stored by the caller, and at step 617 stores the message with the preamble in the mailbox. If at test 615 the VMS determines that the EAP had not passed any preamble data, then the caller's message is stored at step 617 without a preamble. The VMS then returns control of the session to the EAP at step 618. The visit diagrammed in steps 612-618 is logically identical to the carrying out of any command (cf., step 607), and so after returning control to the EAP at step 618, the VMS sends a response to the EAP at step 608, and the session continues from there as described above.

If at test 605 the command is not a visit command but at test 606 it is a transfer command, then at step 619 the VMS stores the logical channel identifier, the session identifier, the application type, and a return location, so that if the transfer is unsuccessful, the VMS can return control to the EAP and the session can continue. At step 620, the VMS rings the desired extension and tests at test 621 to see whether or not the extension is answered. If it is, then at test 622 the VMS tests to see whether or not the EAP has passed any introduction information. If the VMS has received introduction information, then at step 623 the VMS plays the introduction information to the extension, and then connects the caller at step 624. If at test 622 there is no introduction information, then at 624 the caller is connected directly without an introduction. The VMS then sends a completion response to the EAP at 608, as above. Once the caller is connected to a human being, the VMS loses control of the call. Therefore, in this case the determination made by the EAP at test 609 is not to continue the session. At step 610, the call is identified as having been transformed to the extension and the session ends at 611. While the recipient of the call may transfer the caller back to an IVR application, any information previously entered will have been lost, and a new session would begin.

If at test 621 the extension is not answered after some predetermined interval, then an appropriate response is sent to the EAP at 608, and the EAP determines at 609/610 whether or not to connect the caller to the voice mailbox associated with the called extension. This decision is determined by the EAP programming, which in turn depends on the VMS configuration and the level of PBX/central office integration. If, at 609/610, the EAP determines that the caller should be connected to the mailbox, then as at step 613 the VMS transfers the caller to the voice mailbox associated with the called extension. From there, the session continues as an IVR session, using the data stored in step 619. Thus after leaving message for the intended recipient of the call transfer, the caller may perform other functions, or may decide to end the session by hanging up or by entering the appropriate keystrokes. If at 609/610 the EAP determines that the caller should not be connected to the mailbox, then the EAP ends the session or executes one of the other alternatives at step 610,

usually by playing a menu to the caller.

It is understood of course that a caller can disconnect from the system at any time by hanging up or entering an end-of-session command, or may be disconnected by a switching error. In such a case, as long as step 624 has not been reached, the session continues to the point that the VMS responds to the EAP at step 608. In this case the response would indicate that the caller hung up or entered an end-of-session command, and the likely result is that at test 609 the system would terminate the session. However, before doing so it might send a message to a predetermined voice mailbox (e.g., that of the intended call recipient in a transfer, or some designated system mailbox that is regularly emptied by an operator) reporting that the caller, if sufficiently identified, or a caller, if there was insufficient identification, had called.

Thus it is seen that an integrated voice messaging/interactive voice response system in which information given by a caller could be transferred among various application modules of the system is provided. One skilled in the art will appreciate that the present invention can be practiced by other than the described embodiments, which are presented for purposes of illustration and not of limitation, and the present invention is limited only by the claims which follow.

CLAIMS: What is claimed is:

[*1] 1. An interactive voice processing system comprising:

interface means for connecting said interactive voice processing system to one or more incoming telephone lines, through each of which a caller engages in a respective call session with said system;

a plurality of voice processing modules, each performing a respective voice processing feature; and

a voice response processor linking said interface means to said voice processing modules; wherein:

at least one of said voice processing modules gathers data from said caller;

said caller transfers among said voice processing modules; and

said voice response processor passes at least a portion of said gathered data from said at least one of said voice processing modules to other of said voice processing modules when input by said caller results in a transfer among said voice processing modules.

[*2] 2. The interactive voice processing system of claim 1 further comprising a host computer, wherein at least one of said voice processing modules is a voice response module connected to said host computer for processing data entered into said voice response module by said caller.

[*3] 3. The interactive voice processing system of claim 2 wherein, in addition to said at least a portion of said gathered data, data related to said caller and previously known to said host computer are passed to said other of said voice processing modules.

[*4] 4. The interactive voice processing system of claim 2 wherein:

said host computer supports a database;

said data entered by said caller comprise queries of said database; and

said voice response module formats said data entered by said caller into query messages recognizable by said host computer, transmits said query messages to said host computer, receives responses to said query messages from said host computer, and provides voice responses to said queries to said caller.

[*5] 5. The interactive voice processing system of claim 1 further comprising means associated with said voice response processor for informing a caller, when said caller transfers from one of said voice processing modules to another of said voice processing modules, that said at least a portion of said gathered data has been transferred to said another of said voice processing modules.

[*6] 6. The interactive voice processing system of claim 1 wherein:

at least two of said voice processing modules are audio response modules;

when said caller is connected to a first of said audio response modules, said first audio response module collects audio data from said caller; and

when said caller transfers to a second of said audio response modules, said voice response processor passes at least a portion of said audio data to said second audio response module.

[*7] 7. The interactive voice processing system of claim 6 wherein:

said first audio response module is a voice form module and said collected audio data includes the name of said caller;

said second audio response module is a call transfer module for transferring said call to a person; and

when said caller is transferred to said person by said call transfer module, a portion of said collected audio data including at least said name is announced to said person before said caller is connected to said person.

[*8] 8. The interactive voice processing system of claim 6 wherein:

said first audio response module is a voice form module and said collected audio data includes the name of said caller;

said second audio response module is a voice messaging module for recording voice messages in electronic mailboxes assigned to particular individuals; and

when said caller is transferred to one of said electronic mailboxes by said voice messaging module, a portion of said collected audio data including at least said name is entered into said mailbox as an introduction before said caller is connected to said mailbox.

[*9] 9. The interactive voice processing system of claim 1 wherein:

at least two of said voice processing modules are digital response modules;

when said caller transfers to a first of said digital response modules, said first digital response module collects digital data from said caller; and

when said caller transfers to a second of said digital response modules, said voice response processor passes at least a portion of said digital data to said second digital response module.

[*10] 10. The interactive voice response system of claim 1 wherein:

said incoming telephone lines are directed to said system by one of (a) a telephone company central office and (b) a private branch exchange;

one or more of said incoming telephone lines are assigned to particular destinations;

said system is integrated with said one of (a) said telephone company central office and (b) said private branch exchange, such that said system associates an incoming call with one of said destinations; and

said voice response processor passes said destination to at least one of said voice processing modules when input by said caller results in a transfer among said voice processing modules.

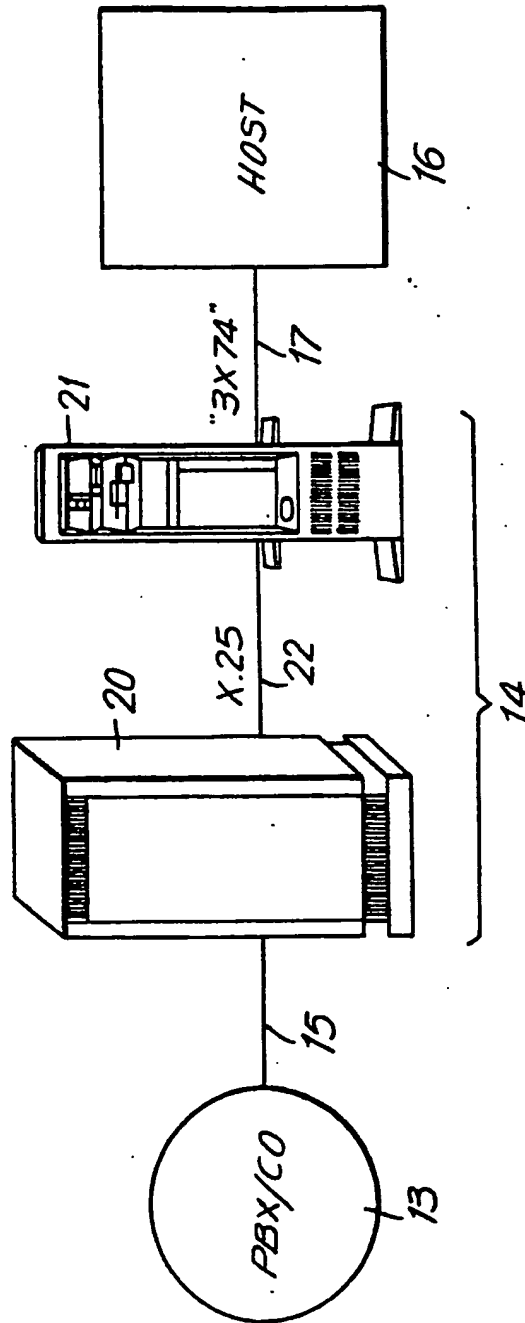
U.S. Patent

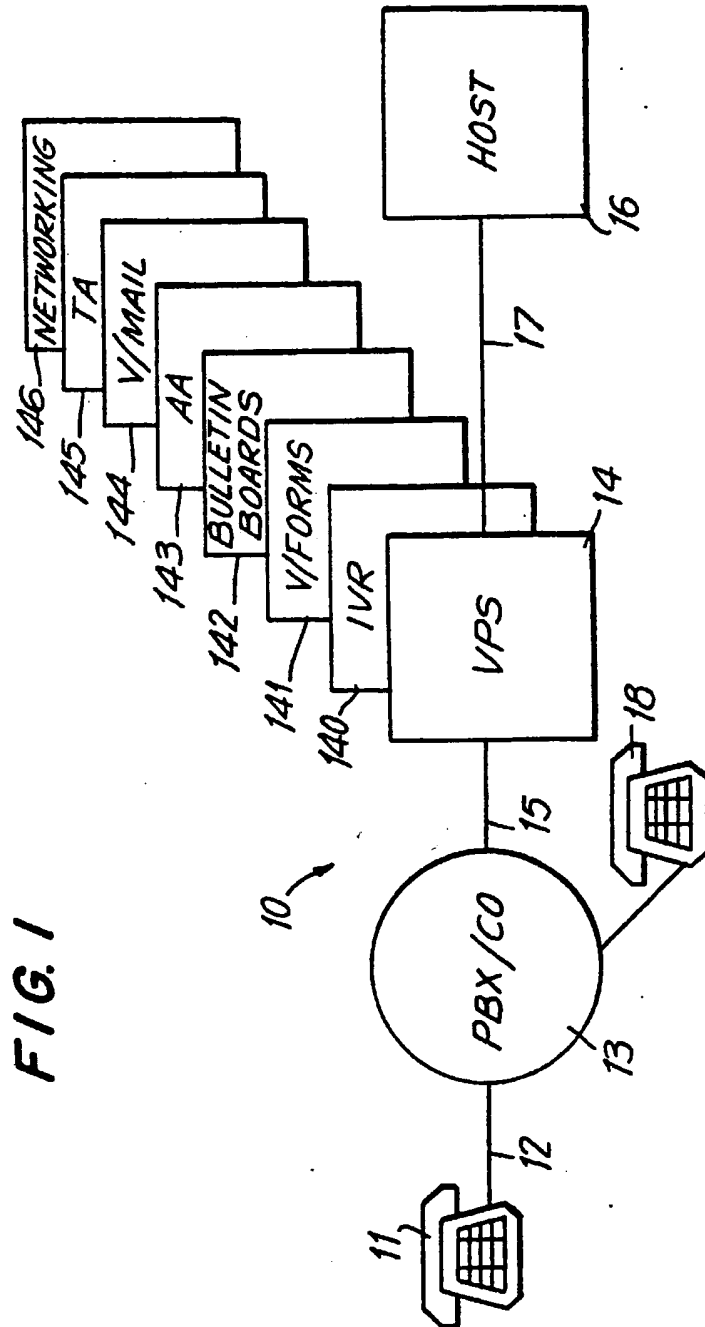
Jan. 12, 1993

Sheet 2 of 7

5,179,585

FIG. 2





U.S. Patent

Jan. 12, 1993

Sheet 3 of 7

5,179,585

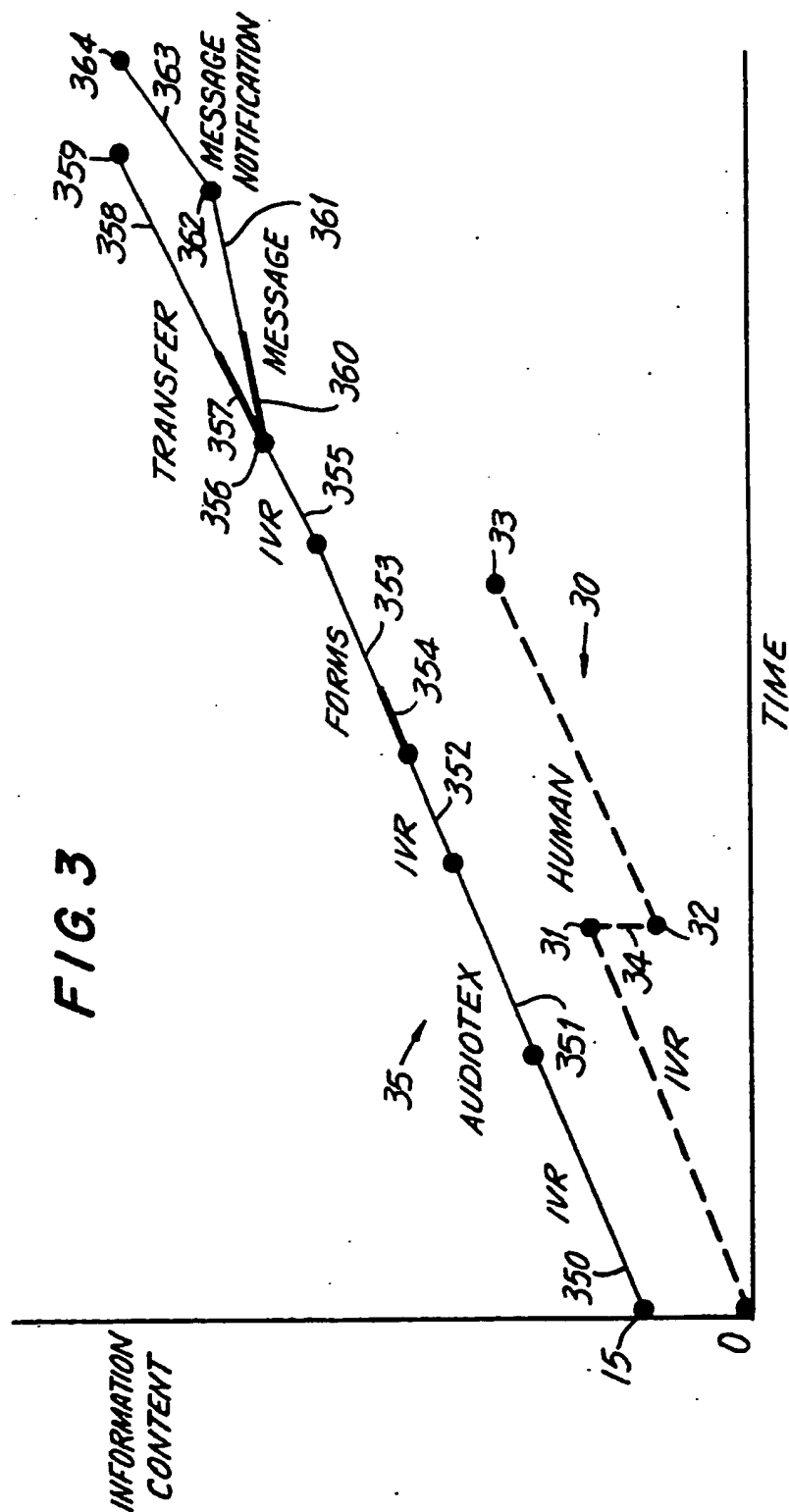


FIG. 4

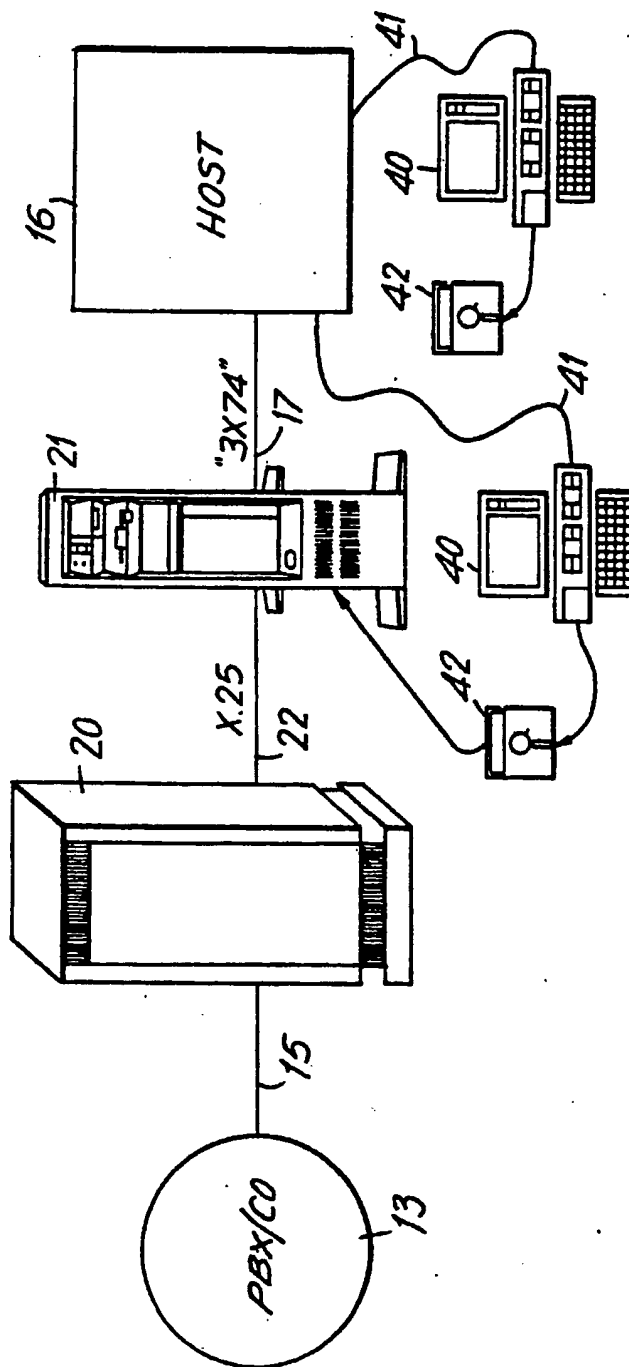


FIG. 5

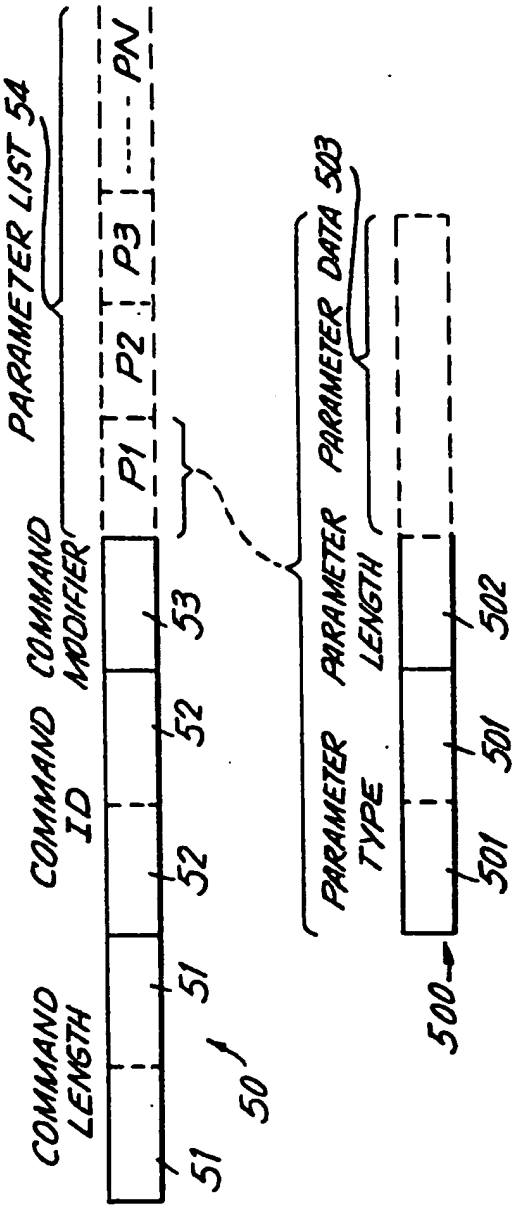


FIG. 6A

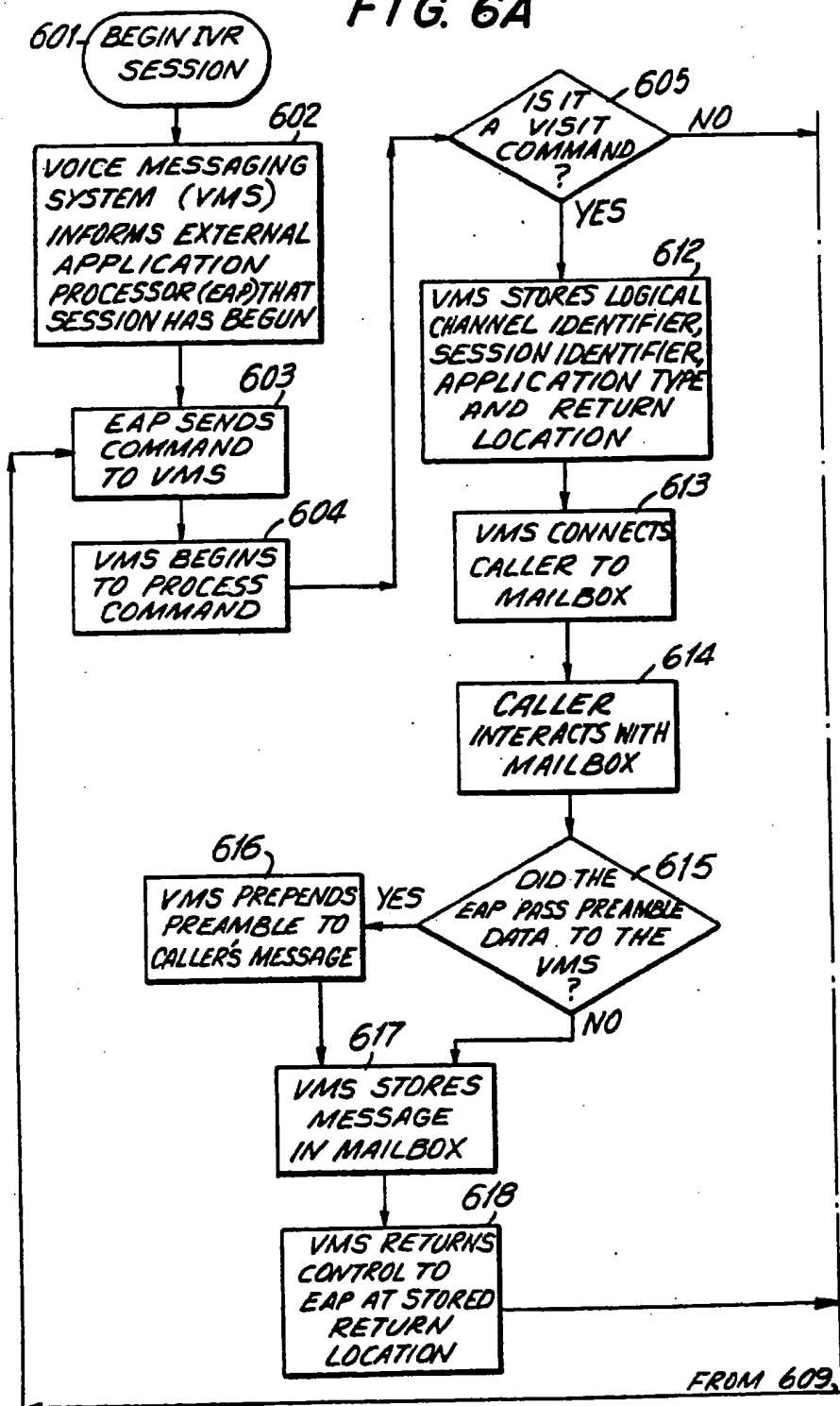
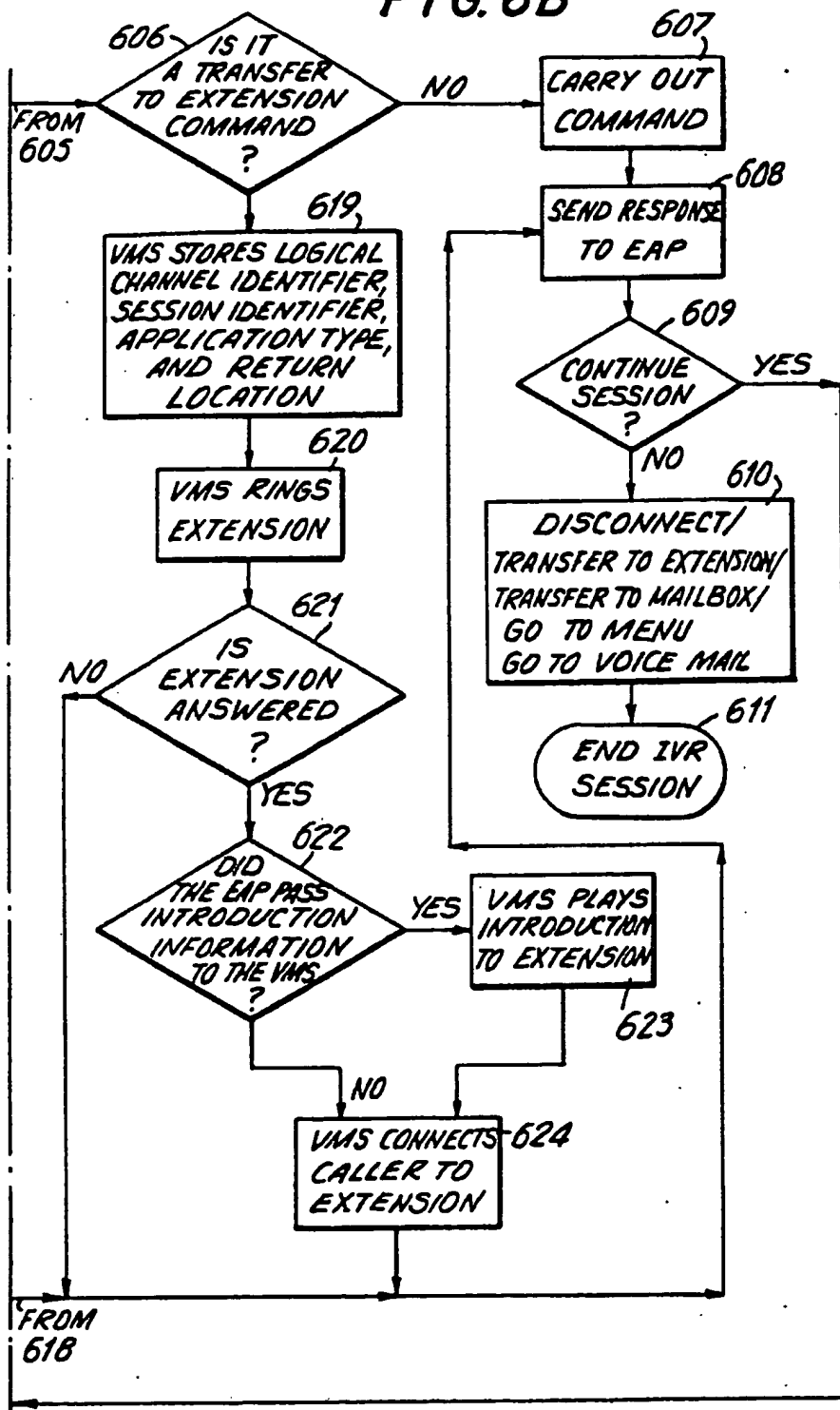


FIG. 6B



7

Pat. No. 5572570 printed in FULL format.

5,572,570

<=2> GET 1st DRAWING SHEET OF 2

Nov. 5, 1996

Telecommunication system tester with voice recognition
capability

LIT-REEX: NOTICE OF LITIGATION

Filed Jan. 25, 2000, D.C. Massachusetts, Doc. No. 00CV10150 (PLAINTIFF AND
DEFENDANT UNAVAILABLE)

NOTICE OF LITIGATION

Interactive Quality Services v. Teradyne, Inc., Filed Apr. 8, 1997, D.C.
Minnesota (Minneapolis), Doc. No. 97-842 JMR/FLN

INVENTOR: Kuenzig, John F., North Chelmsford, Massachusetts

ASSIGNEE-AT-ISSUE: Teradyne, Inc., Boston, Massachusetts (02)

ASSIGNEE-AFTER-ISSUE: Date Transaction Recorded: Aug. 08, 1997
ASSIGNMENT OF ASSIGNOR'S INTEREST (SEE DOCUMENT FOR DETAILS).
HAMMER TECHNOLOGIES, INC. 226 LOWELL STREET WILMINGTON, MASSACHUSETTS 01887
Reel & Frame Number: 8644/0508

APPL-N0: 321,357

FILED: Oct. 11, 1994

INT-CL: [6] H04M 1#24; H04M 3#08; H04M 3#22

US-CL: 379#1; 379#9; 379#10; 379#12; 379#14; 379#15; 379#27; 379#29; 379#32;
379#88.1;

CL: 379;

SEARCH-FLD: 379#1, 9, 10, 12, 18, 14-15, 69, 88, 89, 27, 29, 32

REF-CITED:

U.S. PATENT DOCUMENTS			
4,314,110	2/1982	* Breidenstein	379#18
4,580,016	1/1986	* Williamson	379#31
4,629,836	12/1986	* Walsworth	379#12
5,065,422	11/1991	* Ishikawa	379#18
5,359,646	10/1994	* Johnson	379#27
5,384,822	1/1995	* Brown	379#10

PRIM-EXMR: Chin, Wellington

ASST-EXMR: Shankar, Vijay

CORE TERMS: telecommunication, script, tester, recording, testing, network,

data processing, vocabulary, interface, sampler, recognizer, user, sample, recited, recorded, telephone, routine, tested, detecting, transmitting, reproducing, machine, interactive, substep, automatically, microfiche, appendix, transmitted, detected, message

ABST:

Connection is made directly or via a switched telephone network between a tester and a telecommunication system to be tested. User data, such as voice signals, and signalling data, such as DTMF signals are generated by the tester to cause the telecommunication system to respond. The user and signalling data may be generated interactively to test the telecommunication system in real time, or by executing prerecorded scripts. When the tester is used interactively, the representations of the signals transmitted to and from the tester are recorded in at least one script file. Voice recognition is preferably used to determine whether the response received from the telecommunication system is the expected response, particularly when a script is executed. The vocabulary used for voice recognition may be created by repeatedly executing a script file (recorded during interactive operation of the tester or created manually) which causes the telecommunication system to generate the desired prompt(s) and message(s). Alternatively, the vocabulary may be obtained in other ways, such as by reproducing voice signals at the telecommunication system.

NO-OF-CLAIMS: 30

EXMPL-CLAIM: <=3> 8

NO-OF-FIGURES: 2

NO-DRWNG-PP: 2

SUM:

REFERENCE TO MICROFICHE APPENDIX

A microfiche appendix of 4 fiche having 364 frames is included as part of this application for patent.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a method and apparatus for testing a telecommunication system and more particular, to detecting signals generated by a telecommunication system, such as a voice processing system providing voicemail, or automated attendant, or interactive voice response, etc.; or an automatic call distributor; or a computer-telephone integration system; or command and control components of a switching system; or other networks and enhanced services provided via public or private switched telephone networks, to evaluate the operation, accuracy and quality of the telecommunication system, including machine produced voice signals generated by the system.

2. Description of the Related Art

* For over a century, the public switched telephone network (PSTN) was used primarily to provide voice communication between two human beings using telephones. In the last quarter century or so the PSTN has increasingly been used to provide communication with or between machines, such as computers and facsimile machines. During the last decade and a half, use of the PSTN has expanded to include communication between human beings using telephones and machines which reproduce voice signals that have been previously recorded. The term telecommunication system will be used to refer to equipment which performs voice processing, such as voicemail, automated attendant, interactive voice response (IVR), etc. In addition, the term telecommunication system as used herein includes systems which provide computer-telephone integration (CTI), automatic call distribution (ACD) and many other functions in addition to the more traditional functions of telecommunication systems such as those provided by private branch exchanges (PBXs), central office (CO) equipment, etc.

Prior to the development of new telecommunication functions in the last two decades or so, telecommunication systems were primarily tested by establishing that a voice conversation could be carried from point A to point B. As a result, telecommunication systems have conventionally been tested to determine that connections are made properly and signal quality is satisfactory. While both proper connections and signal quality still need to be tested, the additional functions performed by many different types of telecommunication systems require that a much different type of testing be performed. Initially, human beings were used to test telecommunication systems having voice processing capability.

The assignee of the present invention introduced the Hammer TM testing and monitoring system in 1992. The Hammer TM system uses a personal computer with voice processing capability to execute "programs" written in a language similar to BASIC with extensions for control of the voice processing and telecommunication capabilities of the computer. The extensions include the ability to play prerecorded voice messages, generate dual tone multifrequency (DTMF) and multifrequency (MF) signals for controlling telecommunication systems, such as voicemail systems and switching systems in the PSTN, respectively. In addition, an RS-232 serial port provided in the Hammer TM system can transmit and receive out-of-band signals. As a result, the previously available Hammer TM system can be programmed to test a wide variety of functions provided by telecommunication systems. All that is required is initiation of a stored program by a human operator.

Despite the significant advancements of the Hammer TM system over the totally or near totally manual testing methods that were used before introduction of the Hammer TM system, there are several drawbacks to the previously available Hammer TM system. These drawbacks include the significant amount of time and knowledge required to prepare "programs" to perform automatic testing of telecommunication systems; the need to create each "program" prior to performing testing (batch processing only); the need for human beings to initiate execution of each "program"; and the need for human beings to evaluate the voice signals generated by telecommunication systems under test. No other known test system provided these features prior to the development of the present invention.

SUMMARY OF THE INVENTION

An object of the present invention is to support the gathering, management and quality control of speech and data samples from the speech and data

interfaces of a telecommunication system running any telephony application which includes a speech or data interface.

Another object of the present invention is to reduce human involvement required during testing of telecommunication systems, particularly systems which automatically generate voice signals.

A further object of the present invention is to simplify the generation of "programs" for controlling telecommunication system test equipment.

A still further object of the present invention is to enable testing a telecommunication systems at any time without requiring initiation of testing at that time by a human being.

Another object of the present invention is to provide computer controlled test equipment capable of testing telecommunication systems interactively, particularly over a switched telephone network.

Yet another object of the present invention is to improve the collection of data concerning operation of a telecommunication system under test.

A still further object of the present invention is to provide automatic voice recognition of responses generated by a telecommunication system under test.

These objects are attained by providing a method of testing a telecommunication system including generating, in a data processing device, test signals simulating at least one of the user data and signaling data transmissible by a telecommunication network; transmitting the test signals to the telecommunication system; recording representations of the test signals in the data processing device during the generation or transmitting of test signals; and detecting response signals produced by the telecommunication system in response to the test signals. The user data will often constitute voice signals, but could also include machine generated signals, such as those generated by a facsimile machine, which are treated as a message from a user of the telecommunication system rather than being used to control the telecommunication system. The signaling data includes nonvoice signals, such as DTMF signals which are detected by the telecommunication system to change the service provided a user of the system, as well as other in-band control signals, such as multifrequency signals used to control switching of telecommunication systems and out-of-band signals, such as simplified message desk interface (SMDI), automatic number identification (ANI) and dialed number identification services (DNIS). Also included in the signaling data are signals such as the reversal of polarity or "winking" of telecommunication signal lines. Timing information is also recorded.

According to the present invention telecommunication systems are tested using a data processing device controlled by a processor having capabilities similar to that provided by an INTEL 80386, or preferably 80486 microprocessor connected to telecommunication interface devices capable of generating DTMF and other in-band signals in addition to voice signals and preferably including voice recognition capability. In a preferred embodiment, voice recognition is provided by a peripheral device connected to the microprocessor and software primarily executing on that peripheral device. The data processing device also includes sufficient main memory and long-term storage (currently typically provided by hard disk drives) to store the operating system, test application programs and

"scripts" executed by the microprocessor to test complex telecommunication systems. Also included is an audio output device, such as a conventional speaker, or headphone (not shown).

A data processing device used, according to the present invention, for testing telecommunication systems is programmed to permit interactive generation of the test signals (both voice and non-voice) while connected either directly, or via the PSTN, to the telecommunication system under test. Connections may include a serial line connection to enable generation and detection of out-of-band signals by the data processing device.

During interactive generation of the test signals, a sampler script containing representations of the test signals and a recognizer script containing representations of both the test signals and responses thereto, are stored in the long-term memory of the data processing device. Either of these scripts can be edited and reproduced with or without editing, to perform further testing of the telecommunication system under test. The sampler script may be used to generate a plurality of samples of voice signals produced by the telecommunication system. These samples can, in turn, be used to build a vocabulary for use by the voice recognition equipment in the data processing device. The recognizer script, after editing if necessary, can be used to test the telecommunication system to determine whether subsequent voice response signals match the expected voice response signals. These capabilities enable the present invention to be used for in-service testing when the telecommunication system is available for serving calls from users, as well as in a laboratory environment.

These objects, together with other objects and advantages which will be subsequently apparent, reside in the details of construction and operation as more fully hereinafter described and claimed, reference being had to the accompany drawings forming a part hereof, wherein like reference numerals refer to like parts throughout.

DRWDESC:
BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a data processing device for testing telecommunication systems according to the present invention;

FIG. 2 is an example of the data processing device illustrated in FIG. 1 connected to a telecommunication system for purposes of testing the telecommunication system.

DETDESC:
DESCRIPTION OF THE PREFERRED EMBODIMENTS

Previously, it was sufficient to simply qualify the transmission quality of telecommunication networks, but now it has become necessary to qualify the characteristics of the automated system answering the phone in order to ensure customer satisfaction. This is a far more complex test task and telecommunication test technology has not kept pace with this evolution in requirements.

To adequately monitor these automated systems, both the application itself and the quality of the transmission media between the caller and the system answering the phone must be checked. The test tool must both provide the normal inputs provided to the system being tested, as well as monitor the responses from the system being tested.

The possible responses of a system being tested include signalling data and user data, typically voice signals (i.e., something recognizable by the human ear as representative of human speech), or machine data (i.e., something that a machine can recognize but the human ear cannot) which may or may not be presented in the range of normal human hearing. To measure customer satisfaction, these responses must be measured in terms of time, frequency, and quality.

Present-day telephony applications typically reside on automated systems composed of one or more subsystems, both hardware and software modules, each of which must be tested independently of each other to ensure proper interoperability. In the process of development, it is important to provide "real world" test conditions as early as possible in the product lifecycle. Once each new application has been developed and deployed, the application must be monitored to ensure proper use and function.

To test the entire application in the development phase requires testing from the end-user's perspective, simulating an actual caller's inputs, and measuring the application's responses. In many cases this requires providing many hundreds of possible user inputs to the embryonic application, and measuring many hundreds of possible responses. It is therefore of paramount importance that the test system support easy data collection mechanisms, and support easy management of test suites as well as their results.

The internal architecture of modern-day telecommunication systems is typically very complex, and the traditional test methods are not effective. The traditional link or "continuity" testers will indicate everything is okay after the target system has picked up the phone line. The system could say: "I'm sorry, please call again in 5 years" and the traditional test methods would indicate a satisfactory result if the signal level of that message met predefined parameters.

The present invention determines that a telecommunication system's speech interface is accurate by training the test system to recognize the telecommunication system's speech interface. Factors which must be taken into account include the changing nature of the phone lines connecting the typical caller to the application, the variance in the application due to different loading conditions, and a multitude of other considerations. Thus, it is important to capture samples of the speech interface in an environment as close as possible to that which the system will see in actual use. Speech recognition of the target system's voice processing interface might work very well in a laboratory environment, but not work well at all after the application is deployed, if the samples gathered in the laboratory are not representative of the real world conditions experienced in the field.

Having captured potentially thousands of speech samples, the present invention is able to easily capture additional samples of the same speech segments from the application running on the telecommunication system being tested, to allow for differences in operating conditions. The present

invention provides tools that are easy to use and modify to support the capture and later recognition of speech samples gathered from the system being tested. In addition, subsequent management of the testing process is provided along with automation tools which help the test designer handle the profusion of speech interfaces that are likely to be encountered.

As illustrated in FIG. 1, a data processing device according to the present invention includes the components illustrated in FIG. 1. A processor 20, such as a microprocessor having the capabilities of an INTEL 80386 or preferably i486, is connected to a number of other components via a bus 22, such as an industry standard architecture (ISA) bus. These components include the conventional components of a personal computer, such as a display 24 and keyboard 26 or other input device, such as a touch screen, digitizer pad, pen input device, mouse, etc. In addition, random access memory (RAM) 28 provides system memory and may be connected via the bus 22 or a separate bus to the processor 20. Read only memory (ROM) 30 typically stores basic input/output system (BIOS) programs to control operation of the computer system. Also, some type of memory is provided for long-term storage, such as one or more disk drives 32, e.g., hard disk drives and floppy disk drives, or tape drives, flash memory, bubble memory or any other technology for non-volatile storage.

A data processing device according to the present invention includes a number of interfaces for communication with external devices. Conventional serial 34 and parallel 36 interfaces are provided for connection to printers and conventional communication devices such as modems. In addition, at least one serial interface 34 is used by a data processing device according to the present invention to communicate with telecommunication systems using the SMDI protocol. Unlike conventional personal computers, a data processing device according to the present invention also includes one or more telecommunication network interfaces 38 which can be connected to telecommunication equipment in the same manner as a standard telephone or conventional telephone network equipment, such as a PBX. Examples of telephone network interface cards include models DTI/211, DTI/101 and LSI/120 manufactured by Dialogic Corporation of Parsippany, N.J. Voice digitization units 40 provide the ability to store and output voice signals. Examples of voice digitization units include models D121B and D41D available from Dialogic Corporation. To enable the data processing device to recognize voice signals from a telecommunication system under test, speech recognition units 42, such as Vpro42 or Vpro84 from Voice Processing Corporation of Cambridge, Mass. are provided.

To enable the data processing device illustrated in FIG. 1 to operate according to the present invention, programs are stored in the long-term storage 32 and loaded into the RAM 28 for execution. In one embodiment of the present invention, these programs include a UNIX operating system from Santa Cruz Operation, Inc. of Santa Cruz, Calif., the C language routines in the microfiche appendix, and software provided by Dialogic Corporation and Voice Processing Corporation to interface with the interface(s) 38 and units 40 and 42. Additional C language routines which can be provided by an ordinarily skilled C programmer are also stored in the long-term storage 32. While the C language is used in the disclosed embodiment, as known in the art, the same functions could be provided by a wide range of data processing systems, from other microprocessor controlled computers to supercomputers, executing programs written in any language or combination of languages which can provide control of components capable of connection to a telecommunication system.

When the contents of the microfiche appendix are used to control the data processing device illustrated in FIG. 1, the main routines are in the channel.c file. These routines provide the ability to control the interface(s) 38 and units 40 and 42 to test a telecommunication system connected to the data processing device by playing scripts written in a language similar to BASIC with commands that control the interface(s) 38 and units 40 and 42. The commands in the language implemented using the programs in the microfiche appendix appear in the case statements on pages 19-84 of channel.c in the channel-main routine.

A user controls testing of the telecommunication system through the execution of the routines in the console.c file. The routine do-console-mode receives input from the keyboard and interfaces with the interface(s) 38 and unit(s) 40 through an interface to the routines provided by Dialogic Corporation. The routine disp-status-line displays a status line on the display 24 indicating how the telecommunication system is connected.

There are several ways in which the data processing device illustrated in FIG. 1 can be connected to a telecommunication system to be tested. In FIG. 2 dashed and dotted lines are used to illustrate alternative connections between a tester 50, which may be constructed as illustrated in FIG. 1, and a telecommunication system to be tested. The standard telephone or POTS 52 would be connected via a standard analog line 54. The tester 50 may be connected to a switched telephone network 56 via an analog line 58, digital line 60, or other connection 62. The tester 50 may also be more directly connected to a local user switch 64 or voice response unit 66 by any connection supported by the switch 64 or unit 66. The routines in make-conn.c establish the protocol used by four different connections under the control of make-connection: (1) a standard analog connection; (2) a call driver connection for connecting the tester 50 to several POTS connections on the telecommunication system being tested; (3) a ringdown connection via a ringdown box, such as a model AS-66 from Skutch Electronics of Roseville, Calif.; and (4) a digital connection, e.g., via a T1 line.

In the preferred embodiment, the programs executed by the tester 50 provide the capabilities of soundseer.c in the microfiche appendix. The routines in console.c are called by soundseer.c to provide an interactive interface in real time between a human operator of the tester 50 and the telecommunication system under test. The operator can instruct the tester 50 to select an available channel (pick-available-channel and get-channel); establish a connection with the telecommunication system to be tested (e.g., using get-a-dialer, make-connection, do-analog-connect, do-calldriver-connect, do-ringdown-connect and do-t1-connect); generate DTMF and MF signals (ch-dial) and other signalling data such as winks (ch-wink), flashhook (ch-dial) and on-hook and off-hook (ch-onhook and ch-offhook) using the telecommunication network interface(s) 38; reproduce voice signals or other user data by playing back previously stored files (ch-play); receive signalling data, such as DTMF and MF (ch-getdig); wait for energy (ch-wtnsil) or silence (ch-wtfsil) on line; and record samples of user data, such as voice signals generated by the telecommunication system (ch-record).

In the preferred embodiment codes are stored in a temporary file representing everything generated by the tester 50 during each interactive session. Preferably a separate file is created with representations of the response(s) by the telecommunication system under test. In the remainder of the description of the invention, the former file will be referred as a sampler script and the

latter file will be referred to as a recognizer script. The signals transmitted to and from the tester 50 sampler and recognizer scripts may be recorded in the temporary files by routines like add-to-sampler-script and add-to-recognizer-script, respectively, in the backend.c file in the microfiche appendix.

While it would be possible to record only the recognizer file and edit the file after a session to obtain the contents of the sampler file, in many cases both files are needed and so it is preferable to record two temporary files and have the operator indicate which one(s) to save permanently. When the operator of the tester 50 leaves the real time interactive mode, the operator is preferably prompted to save each of the sampler and recognizer scripts (do-console-mode calls save-test-script in console.c).

The information recorded in the sampler script includes representations of signalling data, such as DTMF signals, MF signals, changes in polarity ("winking" of analog line), etc. Also stored is timing information regarding when the signalling data is generated and when a response is expected from the telecommunication system under test. Information identifying prerecorded files transmitted to the telecommunication system under test is also recorded. In addition, changes to the status of the tester 50 are also recorded. For example, when the tester 50 executes the programs in the microfiche appendix, there are several toggled states, such as DTMF/MF signal generation, which are recorded as comments in the sampler and recognizer scripts.

The recognizer script records everything in the sampler script and also records representations identifying what was received by the tester 50 from the telecommunication system. Under certain known and common circumstances the output from the tester 50 provides a stimulus which is expected by the operator to elicit a particular response from the telecommunication system under test. The expected response is indicated by the operator after an instruction is input to produce the stimulus. For example, the operator may press "1" to enter a voice mailbox number on the system under test and then press "r" to instruct the tester 50 to expect a response from the system, i.e., to perform a recognize command.

In executing the recognize command, the tester 50 will accept one or more of DTMF, MF and voice signals as valid responses, depending on the current state of the internal toggle "collection-type" which is found in the structure "session" defined in the soundseer.h file in the microfiche appendix. If the current collection-type is DTMF, the sampler script would contain an instruction to gather DTMF digits from the system under test. The recognizer script would, in addition to this instruction, contain instructions to save the digits collected, and compare them to the digits collected in the interactive session. Likewise, if the collection-type was voice, the sampler script would contain instructions to gather machine generated speech signals and the recognizer script would also contain instructions to get a speech recognition unit 42 to use, tell the speech recognition unit to listen to the line connected to the telecommunication system under test and try to determine what was heard, and compare that with the expected response. Thus, where the sampler script only causes the stimuli/response pattern to happen, the recognizer script determines whether the response was the proper and expected one, and provides an indication of success or failure of the telecommunication system to generate the expected response.

In the preferred embodiment, the operator is able to replay the sampler and recognizer scripts during interactive operation of the tester 50, or at a time in the future scheduled by a command from the operator. An operator of the tester 50 may edit either the sampler or recognizer script prior to being replayed using a conventional text editor.

One of the uses of the sampler script is to obtain several samples of the same segment of machine generated speech or voice response signals from the telecommunication system for use by the speech recognition unit(s) 42. As discussed above, during interactive use an operator can issue a command from the keyboard to save voice signals, received from the telecommunication system under test, in a file. The instruction to store voice signals in a file would be saved in the sampler script. If no instruction is issued during the interactive session which created the sampler script, the sampler script may be edited to add instruction(s) at the desired point(s).

The speech recognition unit(s) 42 use vocabulary files to recognize speech. Vocabulary files may be created (using `ch-create-disc-vocab` if using the programs in the microfiche appendix) and modified (`ch-load-disc-vocab`). The operator of the tester 50 can then listen to and name a sample file (`play-and-name-sample`) and, if desired, add that sample to the current vocabulary (`save-to-vocab-name`). The new sample will then be added to the current vocabulary (`ch-add-pronunciation`) in the event the operator wishes to save the vocabulary later. The speech recognition routine names with the prefix "ch-" can be found in the `srec.c` source file and other routines can be found in the `vocab-utils.c` source file.

In the preferred embodiment routines in `build-vocab.c` in the microfiche appendix can be used to automatically create vocabulary files. A new vocabulary is created (`-build-new-vocab`) the samples are named (`-rename-existing-files`) and moved to a new vocabulary location (`-move-vocab-files`). The routines `load-vocab-from-samples` and `save-vocab` in the `vocab-utils.c` source file are called by the routines in `build-vocab.c` to perform these functions.

After creation, either manually or automatically, a vocabulary may be edited by adding words as described above, by playing an existing sample from the vocabulary (`ch-play`) and by deleting a word in the vocabulary (`remove-word-from-vocab`). The modified (or newly created) vocabulary may be saved for permanent use (`ch-create-disc-vocab` and `save-vocab`). All samples in the new vocabulary are preferably loaded into a speech recognition unit 42 at this time (`load-vocab-from-samples`).

The vocabulary can be stored for use by any test script, including recognizer scripts, but also including test scripts generated manually. Similarly, the samples of speech can be obtained using sampler script(s) or any other means, such as from an operator of the telecommunication system under test. If subsequent testing via the PSTN (e.g., in service testing) is to be performed, it is generally preferable to obtain the speech samples when the tester 50 is connected via the PSTN, since signal quality can vary.

An example of how the present invention can be used to test a telecommunication system will be provided with reference to FIG. 2. In normal operation, a caller on telephone 52 dials a telephone number which the switched telephone network 56 determines should be connected to local user switch 64. As indicated in FIG. 2, the switched telephone network 56 may be the PSTN or a

private network. The local user switch 64 connects the telephone 52 to voice response unit 66. Voice response unit 66, for example, may request data from the caller by playing a prompt, such as "Please enter your credit card number." After receiving the desired information from the caller, e.g., in the form of DTMF signals or a voice command, voice response unit 66 may generate another prompt, such as "To check an account balance, select 1; to talk to a customer service representative, select 2." IF the caller requests an account balance, the voice response unit 66 issues a request for the account balance over data link 68 to host or file server 70. If an account balance is received, the voice response unit 66 will convert the account balance into voice signals. However, if the account balance is not obtained due to a failure in the data link 68, host or file server 70 or voice response unit 66, the voice response unit will generate a message such as "Your account balance is not currently available, please try again later."

] # 25

In the example illustrated in FIG. 2, the tester 50 is connected in such a manner that in-service testing may be performed. In other words, callers like the one discussed above may be using the telecommunication system 72 at the same time as the tester 50. In addition, multiple calls can be generated by the tester 50 using different ports. This enables the tester 50 to test the telecommunication system 72 under real world conditions with varying amounts of load.

By using voice recognition of the prompts and messages generated by the voice response unit 66, the tester 50 is able to make determinations of what components of the telecommunication system 72 are operating properly. For example, if the initial prompt is received from the voice response unit 66, it can be determined that the local user switch 64 connects to the voice response unit 66. Without voice recognition, the voice signals received could be from the telephone 74 of one of the agents of the organization using the telecommunication system 72 to provide services to callers. A test script can be generated to test a number of different connections that the local user switch is capable of making, using voice recognition and other types of signal detection, e.g., facsimile and modem handshaking protocols, etc.

This type of testing to determine whether all services provided by a telecommunication system are accessible may involve more than a single switch. Present-day telecommunication systems often use a combination of private and commercial networks, such as MCI and SPRINT, to provide services to users. Due to the complexity of such telecommunication systems, it is not uncommon for problems to occur and, on the other hand, it is difficult to perform a complete test of such telecommunication systems without using a tester like the Hammer™, preferably configured to provide the functions of the present invention. A test script can be generated by calling all of the services provided by the telecommunication system 72 while in the interactive mode of the tester 50. The resulting recognizer script, after editing if necessary, can be scheduled to be executed periodically to verify that all of the services are available. The recognizer script will include an instruction to verify that the initial voice signals received from each service which generates voice prompts match the expected greeting. For example the following code may be created in a recognizer script:

```
ENABLE-SPEECH-RECOG
LOAD vocab-name
RECOG-DISCRETE 2000 3000 2000
IF $ $ PHRASE1 ! = expected-prompt
```

SET-FAILURE

ENDIF

These statements will obtain access to one of the speech recognition units 42, listen to the line connected to one of the ports, compare what is "heard" with the "expected-prompt" and inform the operator if what is "heard" does not match. The speech recognition unit(s) 42 in the tester 50 will use vocabulary created from one or more samples of the greeting generated by the service. The samples can be obtained using the sampler script recorded at the same time as the recognizer script or by any other means.

The same test script used to test the local user switch 64 or another test script may be used to test all or a portion of the voice prompts and messages which the voice response unit 66 is capable of generating. The test script could perform a complete test of all of the prompts which can be generated by each application using the voice response unit 66. This is called a regression test. However, since some applications are very complex, the test script could be written to test one user interface at a time, e.g., to test that all of the prompts at a particular level in the logic tree of the application are generated. In the example given above, this would mean verifying that the messages providing the account balance and stating that the account balance is not available, plus any other messages or prompts which may be produced in response to entering the correct account number, are generated under appropriate circumstances.

Another type of test which can be performed by a tester according to the present invention is feature testing of an application provided by a telecommunication system. Unlike the user interface test described above, feature testing emphasizes that when proper responses are provided, the desired service is provided. For example, the tester 50 can be used to test an outdialing feature of a voice mail system. Assuming that a subscriber account has been set up on the host 70 for purposes of testing, a test script can be created which will specify the number to be called (unless this has to be done by the administrator of the telecommunication system 72) which will access one of the ports on the tester 50. The test script can then generate the series of signals, voice or non-voice, as needed to cause the telecommunication system 72 to call the number specified. Using voice recognition, the tester 50 can determine that the port connected to receive calls to the specified number received the expected voice signals generated by the telecommunication system.

The many features and advantages of the present invention are apparent from the detailed specification and thus, it is intended by the appended claims to cover all such features and advantages of the system which fall with the true spirit and scope of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art from the disclosure of this invention, it is not desired to limit the invention to the exact construction and operation illustrated and described, accordingly, suitable modifications and equivalents may be resorted to, as falling within the scope and spirit of the invention.

CLAIMS: What is claimed is:

[*1] 1. A method of testing a telecommunication system, connectable to a telecommunication network, comprising the steps of:

(a) generating, in a data processing device, test signals simulating at least one of voice and non-voice signals transmissible by a telecommunication network;

(b) transmitting the test signals to the telecommunication system;

(c) detecting response signals produced by the telecommunication system in response to the test signals, said response signals including initial voice response signals generated automatically by the telecommunication system;

(d) recording at least one sampler script including representations of only the test signals generated in step (a);

(e) recording at least one recognizer script including representations of both the test signals generated in step (a) and the response signals detected in step (c);

(f) repeatedly reproducing the test signals using the at least one sampler script recorded in step (d);

(g) reproducing the test signals using the at least one recognizer script recorded in step (e);

(h) detecting subsequent voice response signals produced by the telecommunication system in response to the test signals reproduced in step (f);

(i) recording the subsequent voice response signals, wherein said recording in step (i) is repeated to obtain a plurality of voice response files, at least one of the voice response files obtained for each repetition of the test signals reproduced in step (f);

(j) building a vocabulary file, wherein said building of the vocabulary file in step (j) uses the voice responses files recorded in step (i);

(k) detecting subsequent voice response signals produced by the telecommunication system in response to the test signals reproduced in step (g);
and

(l) performing automatic voice recognition on the subsequent voice response signals detected in step (k) using the vocabulary file built in step (j), wherein said detecting in step (k) and the automatic voice recognition in step (l) are performed.

[*2] 2. A method as recited in claim 1,

wherein said reproducing in step (g), said detecting in step (k), and the automatic voice recognition in step (l) are performed repeatedly while the telecommunication system is available to serve calls from users.

[*3] 3. A method of testing a telecommunication system, connectable to a telecommunication network, comprising the steps of:

(a) generating, in a data processing device, test signals simulating at least one of voice and non-voice signals transmissible by a telecommunication network;

(b) transmitting the test signals to the telecommunication system;

(c) detecting response signals produced by the telecommunication system in response to the test signals;

(d) recording at least one sampler script including representations of only the test signals generated in step (a);

(e) recording at least one recognizer script including representations of both the test signals generated in step (a) and the response signals detected in step (c);

(f) executing the at least one sampler script to control the substeps of

(f1) reproducing the test signals recorded in step (d), and

(f2) recording initial voice response signals generated in response to the test signals reproduced in step (f1); and

(g) executing the at least one recognizer script to control the substeps of

(g1) reproducing the test signals recorded in step (e),

(g2) detecting subsequent voice response signals generated in response to the test signals reproduced in step (g1), and

(g3) automatically performing voice recognition on the subsequent voice response signals detected in step (g2) based on the initial voice response signals recorded in step (f2).

[*4] 4. A tester for a telecommunication system connectable to a telecommunication network, comprising:

at least one processor controlling generation of test signals simulating at least one of voice and non-voice signals transmissible by the telecommunication network;

at least one interface, coupling said processor to the telecommunication system, to transmit the test signals and to receive response signals generated by the telecommunication system in response to the test signals, said response signals including voice response signals;

at least one storage device, coupled to at least said processor, storing the test and response signals; and

at least one speech recognition device, coupled to at least said processor, for analyzing the voice response signals.

[*5] 5. A method of testing a telecommunication system including a voice response unit using a data processing device, comprising the steps of:

(a) receiving commands from a human operator of the data processing device to test the telecommunication system in real-time;

(b) generating, in the data processing device, test signals simulating user data and signaling data;

(c) transmitting the test signals to the telecommunication system;

(d) recording representations of the test signals in a sampler and recognizer scripts in temporary storage on the data processing device;

(e) storing representations of response signals produced by the telecommunication system in the recognizer script, including machine produced voice signals from the voice response unit and instructions to perform speech recognition on the machine produced voice signals;

(f) repeating steps (a)-(e) until a command is received from the human operator to step;

(g) prompting the human operator to save the sampler and recognizer scripts;

(h) storing each of the sampler and recognizer scripts in separate permanent files when instructed by the human operator in response to said prompting in step (g);

(i) repeatedly reproducing the sampler script to obtain a plurality of vocabulary samples of the machine produce voice signals from the voice response unit;

(j) automatically building a vocabulary of words from the samples obtained during said reproducing in step (i); and

(k) reproducing the recognizer script to obtain at least one test sample of the machine produced voice signals from the voice response unit and to automatically perform speech recognition on the test sample using the vocabulary built in step (j).

[*6] 6. A method as recited in claim 5, further comprising the steps of:

(l) connecting the data processing device to the telecommunication system via a switched telephone network in response to the commands received in step (a); and

(m) repeatedly performing said reproducing in step (k) during operation of the telecommunication system to serve users.

[*7] 7. A tester as recited in claim 4,

wherein said interface includes an analog connection, a call driver connection, a ringdown connection, and a digital connection, thereby allowing the tester to operate with or without connection to a telecommunication network.

[*8] 8. A method of testing a telecommunication system, the system having a speech interface and being connectable to a telecommunication network, comprising the steps of:

- (a) receiving voice signals produced by the telecommunication system;
- (b) transmitting test signals to the telecommunication system; and

(c) evaluating voice response signals using the voice signals received in step (a), said voice response signals produced by the telecommunication system in response to the test signals transmitted in step (b), thereby determining whether the telecommunication system is operating properly.

[*9] 9. A method as recited in claim 8,

wherein said telecommunication network is a public switched telephone network.

[*10] 10. A method as recited in claim 8,

wherein said receiving in step (a) includes the substeps of:

- (a1) adding representations of said voice signals to a vocabulary file, and
- (a2) optionally editing the vocabulary file.

[*11] 11. A method as recited in claim 10,

wherein said editing in step (a2) includes the substeps of:

(i) listening to at least one of said representations of voice signals in the vocabulary file, and

(ii) optionally deleting the representations of voice signals listened to in step (i).

[*12] 12. A method as recited in claim 10,

wherein said evaluating in step (c) includes the substeps of:

(c1) comparing said voice response signals with said representations of voice signals in the vocabulary file to determine whether the voice response signals match any of the representations in the vocabulary file,

(c2) comparing the representations that match the voice response signals with expected response data to determine whether the telecommunication system produced the proper voice response signals, and

(c3) indicating whether the telecommunication system produced the proper voice response signals in response to the test signals transmitted in step (b).

[*13] 13. A method of testing a telecommunication system, the system being connectable to a telecommunication network, comprising the steps of:

(a) interactively transmitting test signals to the telecommunication system and detecting initial response signals produced by the telecommunication system;

(b) automatically recording representations of both the test signals transmitted and the initial response signals detected in step (a);

(c) reproducing the test signals recorded in step (b);

(d) evaluating subsequent response signals produced by the telecommunication system in response to the test signals reproduced in step (c) to determine whether the telecommunication system is operating properly.

[*14] 14. A method as recited in claim 13,

wherein said initial response signals include voice response signals, and

wherein said interactively transmitting and detecting in step (a) is performed by a human operator who can hear the voice response signals.

[*15] 15. A method as recited in claim 13,

wherein said interactively transmitting and detecting in step (a) comprises the substeps of:

(a1) transmitting test signals to the telecommunication system,

(a2) detecting initial response signals produced by the telecommunication system,

(a3) transmitting test signals to the telecommunication system based on the initial response signals detected in step (a2), and

(a4) repeating steps (a2) and (a3) until the telecommunication system is fully tested.

[*16] 16. A method as recited in claim 13,

wherein said recording in step (b) comprises the substeps of:

(b1) automatically recording at least one file including representations of only the test signals transmitted in step (a), and

(b2) automatically recording at least one file including representations of both the test signals transmitted and the initial response signals detected in step (a).

[*17] 17. A method as recited in claim 16,

wherein said evaluating in step (d) comprises the substeps of:

(d1) comparing said subsequent response signals with said representations of initial response signals recorded in step (b2), to determine whether the telecommunication system produced the proper response signals, and

(d2) indicating whether the telecommunication system produced the proper response signals in response to the test signals reproduced in step (c).

[*18] 18. A method as recited in claim 16,

wherein said reproducing in step (c) comprises the substep of:

(c1) reproducing the test signals recorded in step (b1).

[*19] 19. A method as recited in claim 18,

wherein said recording in step (b) further comprises the substep of:

(b3) automatically adding representations of subsequent response signals to the file recorded in step (b2), said subsequent response signals produced in response to the test signals reproduced in step (c1).

[*20] 20. A method as recited in claim 16,

wherein said reproducing in step (c) comprises the substep of:

(c1) reproducing the test signals recorded in step (b2).

[*21] 21. A method as recited in claim 16,

wherein said recording in steps (b1) and (b2) further includes automatically recording representations of signaling data, timing information, and test status information.

[*22] 22. A method as recited in claim 21,

wherein said representations of signaling data include DTMF signals, MF signals, and indications of changes in polarity.

[*23] 23. A method as recited in claim 21,

wherein said representations of timing information include expected response times, said expected response times including the time when a response signal is expected relative to the time when a test signal is transmitted.

[*24] 24. A method as recited in claim 21,

wherein said representations of test status information include collection type, said collection type selected from the group consisting of DTMF signals, MF signals, and voice signals.

[*25] 25. A method as recited in claim 24,

wherein said collection type is DTMF signals,

wherein said recording in step (b1) further includes recording a first instruction to collect DTMF signals from the telecommunication system, and

wherein said recording in step (b2) further includes recording said first instruction to collect DTMF signals from the telecommunication system, recording a second instruction to save representations of the DTMF signals in said file, and recording a third instruction to compare said representations of the DTMF signals with the representations of the initial response signals.

[*26] 26. A method as recited in claim 24,

wherein said collection type is MF signals,

wherein said recording in step (b1) further includes recording a first instruction to collect MF signals from the telecommunication system, and

wherein said recording in step (b2) further includes recording said first instruction to collect MF signals from the telecommunication system, recording a second instruction to save representations of the MF signals in said file, and recording a third instruction to compare said representations of the MF signals with the representations of the initial response signals.

[*27] 27. A method as recited in claim 24,

wherein said collection type is voice signals,

wherein said recording in step (b1) further includes recording a first instruction to collect voice signals from the telecommunication system, and

wherein said recording in step (b2) further includes recording said first instruction to collect voice signals from the telecommunication system, recording a second instruction to save representations of the voice signals in said file, recording a third instruction to reserve a speech recognition device, recording a fourth instruction to direct said speech recognition device to analyze the representations of the voice signals, and recording a fifth instruction to compare said representations of the voice signals with the representations of the initial response signals.

[*28] 28. A method of testing a telecommunication system using a data processing device, the system being connectable to a telecommunication network, comprising the steps of:

(a) connecting the data processing device to the telecommunication system, without connection through a telecommunication network;

(b) interactively transmitting test signals to the telecommunication system and detecting response signals produced by the telecommunication system, said test signals simulating user data and signaling data; and

(c) evaluating the response signals to determine whether the telecommunication system is operating properly.

[*29] 29. A method as recited in claim 28,

wherein said interactively transmitting and detecting in step (b) comprises the substeps of:

(b1) transmitting test signals to the telecommunication system,

(b2) detecting response signals produced by the telecommunication system,

(b3) transmitting test signals to the telecommunication system based on the response signals detected in step (b2), and

(b4) repeating steps (b2) and (b3) until the telecommunication system is fully tested.

[*30] 30. A method as recited in claim 28,

wherein said evaluating in step (c) comprises the substeps of:

(c1) comparing the response signals detected in step (b) with expected response signals, to determine whether the telecommunication system produced the proper response signals, and

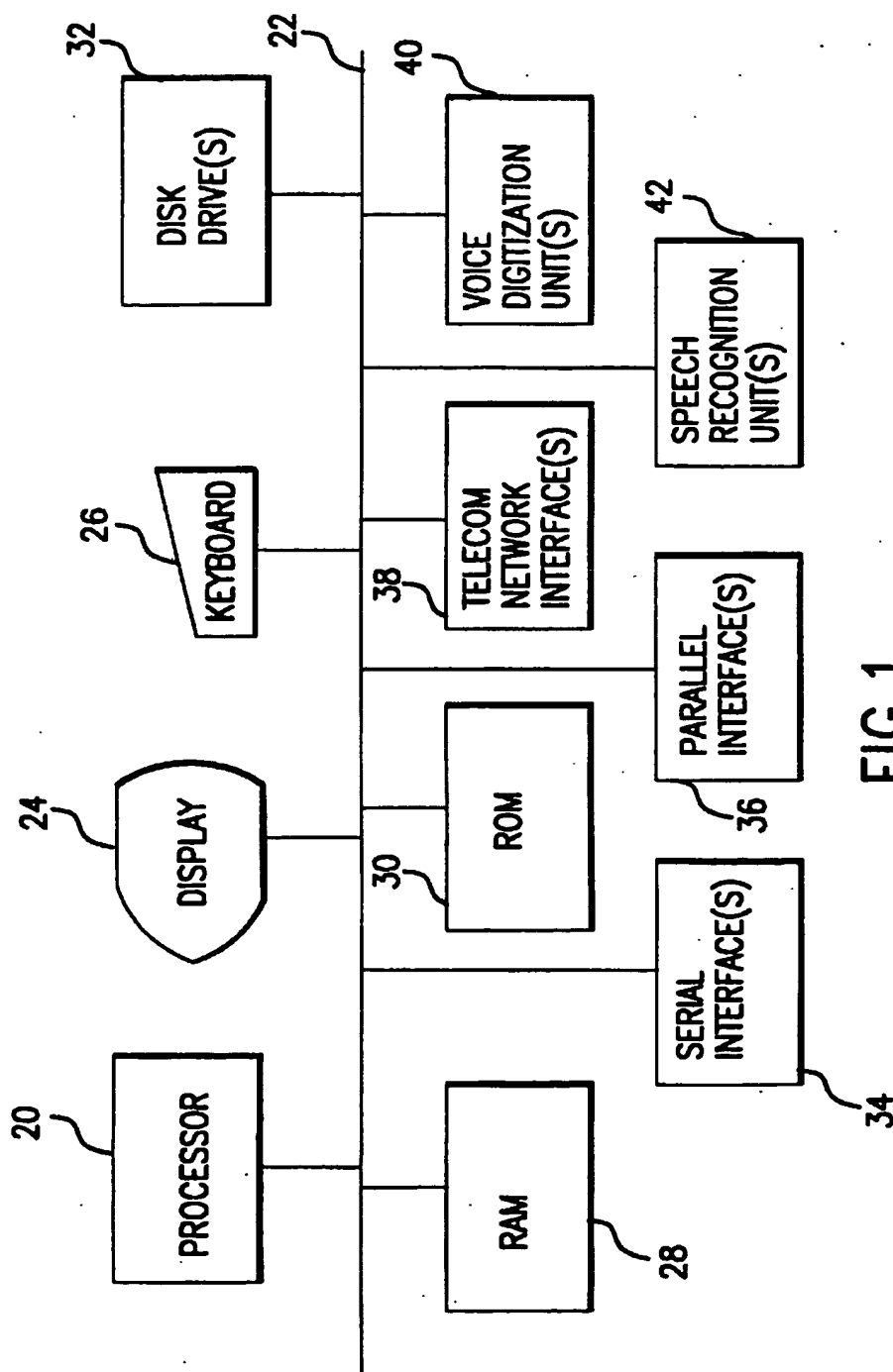
(c2) indicating whether the telecommunication system produced the proper response signals in response to the test signals transmitted in step (b).

U.S. Patent

Nov. 5, 1996

Sheet 1 of 2

5,572,570



U.S. Patent

Nov. 5, 1996

Sheet 2 of 2

5,572,570

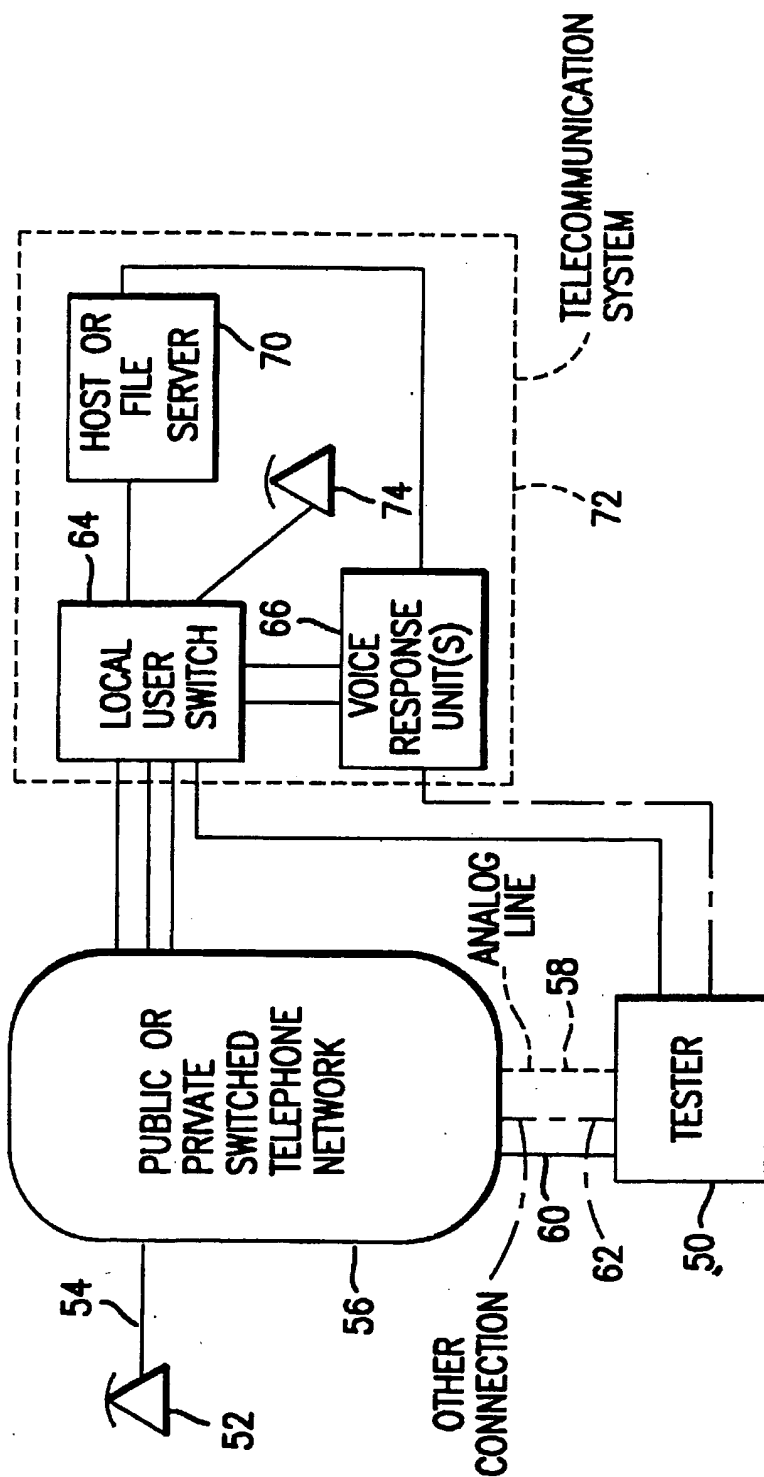


FIG. 2

Pat. No. 5255305 printed in FULL format.

5,255,305

<=2> GET 1st DRAWING SHEET OF 6

Oct. 19, 1993

Integrated voice processing system

INVENTOR: Sattar, Sohail, Irving, Texas

ASSIGNEE-AT-ISSUE: Voiceplex Corporation, Irving, Texas (02)

ASSIGNEE-AFTER-ISSUE: Date Transaction Recorded: Jul. 31, 1995

ASSIGNMENT OF ASSIGNOR'S INTEREST (SEE DOCUMENT FOR DETAILS).

INTERVOICE, INC., A TEXAS CORPORATION 17811 WATERVIEW PARKWAY DALLAS, TEXAS

75252

Reel & Frame Number: 007570/0451

Date Transaction Recorded: Jun. 07, 1999

ASSIGNMENT OF ASSIGNOR'S INTEREST (SEE DOCUMENT FOR DETAILS).

INTERVOICE LIMITED PARTNERSHIP 639 ISBELL ROAD, SUITE 390 RENO, NEVADA 89509

Reel & Frame Number: 009996/0962

Date Transaction Recorded: Jul. 02, 1999

SECURITY INTEREST (SEE DOCUMENT FOR DETAILS).

BANK OF AMERICA NATIONAL TRUST AND SAVINGS ASSOCIATION 66TH FLOOR 901 MAIN

STREET DALLAS, TEXAS 75202-371

Reel & Frame Number: 010095/0518

APPL-N0: 608,147

FILED: Nov. 1, 1990

CERTCORR: Sep. 27, 1994 a Certificate of Correction was issued for this Patent

INT-CL: [5] H04M 1#50; H04M 1#64

US-CL: 379#34; 379#88.9; 379#88.1; 379#93.26; 379#201; 379#216;

CL: 379;

SEARCH-FLD: 379#88, 89, 67, 201, 33, 34, 97, 216

REF-CITED:

U.S. PATENT DOCUMENTS			
4,481,574	11/1984	* DeFino et al.	364#200
4,489,438	12/1984	* Hughes	381#51
4,585,906	4/1986	* Matthews et al.	379#88
4,747,127	5/1988	* Hansen et al.	379#94
4,748,656	5/1988	* Gibbs et al.	379#93
4,755,932	7/1988	* Diedrich	364#200
4,782,517	11/1988	* Bernardis et al.	379#201
4,792,968	12/1988	* Katz	379#92
4,811,381	3/1989	* Woo et al.	379#67

		Pat. No. 5255305, *	
4,850,012	7/1989	* Mehta et al.	379#157
4,893,335	1/1990	* Fuller et al.	379#200
4,922,520	5/1990	* Bernard et al.	379#88
4,926,462	5/1990	* Ladd et al.	379#67
4,930,150	5/1990	* Katz	379#93
5,054,054	10/1991	* Pessia et al.	379#89
5,133,004	7/1992	* Heileman, Jr. et al.	379#67

OTHER PUBLICATIONS

"Conversant 1 Voice System: Architecture and Applications", R. J. Perdue et al, AT&T Technical Journal, vol. 5, No. 5, Sep./Oct. 1986, pp. 34-47.
"Voice Processing Update", Teleconnect, Sep. 1986, pp. 98-138.

PRIM-EXMR: Brown, Thomas W.

LEGAL-REP: Baker & Botts

CORE TERMS: processing, vector, window, user, server, message, telecommunication, database, interface, computer, input, general-purpose, caller, run-time, relational, customer, protocol, recited, integrated, segment, telephone, stored, envelope, digit, stack, exemplary, integration, alarm, label, broadcast

ABST:

Integrated voice processing systems. Generalpurpose common computer platform voice processing systems described provide integrated voice processing functions, for example, voice messaging, call processing, interactive voice response and other systems typically only available in discrete systems. Industry standard computer databases and interfaces are used to create a dynamically modifiable voice processing system which is adaptable to perform to any customer specification. State vector architecture for the system described herein provide economic and efficient tailoring of voice processing functions for a wide variety of applications. Voice processing systems for interfacing voice transactions through a telecommunications line to a user comprise a general-purpose digital, computer common-platform adapted to communicate through the telecommunications line with an outside environment. A relational database interfaced to the general-purpose, digital computer for storing at least one object having a state that is modifiable by a vector protocol, thereby producing a voice transaction event that is output to the user through the telecommunications whereby the user activates the vector protocol through the telecommunications to act on the object and produce the voice transaction event, and an interface adapted to convert user commands input to the voice processing system through the telecommunications to activate the vector protocol and alter the object's state.

NO-OF-CLAIMS: 35

EXMPL-CLAIM: <=19> 1

NO-OF-FIGURES: 19

NO-DRAWNG-PP: 6

SUM:

FIELD OF THE INVENTION

This invention relates to systems for voice processing and methods of providing voice processing functions over telecommunications lines. More specifically, this invention relates to systems and methods for performing integrated voice processing functions and transactions in a general-purpose computer, common-platform environment.

BACKGROUND OF THE INVENTION

Computer-based telecommunications systems have proliferated in the last few years along with the common proliferation of high-speed personal computers and the generally lower costs of equipment now available for use in complex telecommunications applications. With the use of high-speed telephone switching lines, telecommunications applications are exhibiting rapid advancements in technology and versatility. One of the areas in which telecommunications has experienced rapid advancements is the "voice processing" industry, wherein telephone lines provide communication links between users calling in to obtain information from a computer-based system that is adapted to provide information about a particular business or organization.

The voice processing industry provides "voice-based" systems which interact in varying degrees with users seeking information from the system. Voice-based systems have evolved over the last several years into discrete systems which accomplish specific tasks. Thus, the voice processing industry is broken up into a series of sub-industries, each filling niche technologies or "sub-technologies" which are occupied by particular providers and which are further segregated according to the products and services available in the specific sub-technology area. Generally, the voice processing industry has developed the following sub-technology areas: voice massaging ("VM") technology, call processing ("CP") technology, interactive voice response ("IVR") technology, and a number of other limited technologies which at the present are not large and do not command significant market shares, such as for example, the "FAX voice response" technology area. VM systems automatically answer calls and act as "automated attendants" to direct the calls to the proper person or department in an organization. These systems have in the past usually comprised look-up databases that perform voice functions for the user as the user accesses the system. VM technology can be adapted to read electronic mail to a user or caller on a telephone, and may also provide means for storing incoming facsimile messages for forwarding these messages over TOUCHTONE telephones when so instructed. Systems that fall under the VM category may also be adapted to recognize spoken phrases and convert them into system usable data.

Previous VM systems are exemplified in U.S. Pat. No. 4,585,906, Matthews et al. The Matthews et al. patent discloses an electronic voice messaging system which is connected with a user's telephone communications network or private branch exchange (PBX) to provide VM functions for the user. See Matthews et al., col. 4, lines 49-66.

Another example of a VM system is disclosed in U.S. Pat. No. 4,926,462, Ladd et al. The device of the Ladd et al. patent provides methods of handling calls in a VM system based on information supplied by a PBX. See Ladd et al., col.

4, lines 50-52. VM systems taught in the Ladd et al. patent comprise a feature phone emulator interface which emulates known PBX compatible feature phones having multiple line capability. The feature phone emulator is interfaced to the PBX as an actual feature phone, and the PBX is configured to assign a group of extension numbers to line appearances on the feature phone. The VM systems disclosed in the Ladd et al. patent answer the calls to these extensions by using the feature phone emulator interface. See Ladd et al., col. 4, lines 53-65.

Yet another VM system is disclosed in U.S. Pat. No. 4,811,381, Woo et al. The Woo et al. patent VM system which is connected to a trunk side of a PBX in a business telephone system. The VM system described in Woo et al. provides the feature of answering forwarded calls with a personal greeting from the party whose phone is accessed by a user. See Woo et al., col. 2, lines 37 through col. 2, lines 40-54.

If on the other hand a customer requires a voice processing system to perform on-line transaction processing and interact with a caller to answer routine questions about the status of an account, for example, the customer's requirements are usually best addressed by an IVR system which can be viewed as fulfilling requirements presented by a totally different set of architectural problems. Essentially, in an IVR system the user desires to talk to a central processing unit (CPU) to obtain database information. IVRs are particularly useful in the banking industry wherein account holders can call a CPU to get account balances and other relevant information. Generally, IVR systems must also interface to a TOUCHTONE telephone to allow the caller to provide meaningful data to the IVR system which then can return meaningful information to the user.

When a retail company wishes to sell large volumes of merchandise through a "call-in ordering" system, it requires a call processing (CP) system. In the past, before CP systems were available, such retail companies utilized "agents" to handle incoming calls. The agents typically manned a switchboard that allowed manual input of user orders to an ordering system which could have been computer-based. CP technology today provides automatic call distribution (ACD) which allows a company to nearly eliminate the need for live agents handling phone calls, and replaces the agents with an interactive telephone system through which products can be ordered. The products can be paid for by credit cards having credit card numbers which are input through a TOUCHTONE telephone to a computer ordering system for billing purposes.

Other examples of CP technology are taught in U.S. Pat. No. 4,850,012, Mehta et al. The Mehta et al. patent discloses a CP system for intercepting incoming calls to a key telephone system, and returning a message to a calling party. See Mehta et al., col. 2, lines 11-17. The Mehta et al. system further provides an intercom line for providing voice announcements or messages through the key telephone system to the called parties. CP systems described in Mehta et al. comprise a call processor which intercepts telephone calls wherein an instructional message is returned to the calling party, thereby informing the calling party to select a party associated with the key telephone system by dialing a pseudoextension number associated with each party. See Mehta et al., col. 2, lines 18-28.

Other technologies have been developed to provide the particular services and solutions to other niches and subtechnologies in the voice processing industry. Interactive FAX voice processing is a burgeoning sub-technology area and has

required specialized technical advancements to provide efficient voice-activated FAX systems. The technical advancements required to make FAX voice processing and other advanced voice processing systems feasible have not heretofore been adequately developed. There is a long-felt need in the art for a general-purpose system which can effectively, economically, and efficiently provide these technological advancements and which will integrate the above-mentioned other voice-based technologies in the voice processing field.

Examples of such systems for data reception and projection over telephone lines are disclosed in U.S. Pat. Nos. 4,481,574, DeFino et al., and 4,489,438, Hughes. Both the DeFino et al. and Hughes patents teach hard-wired systems which interface to telephone lines and computers to provide telecommunications applications. However, the systems disclosed in the DeFino and Hughes et al. patents generally perform the telecommunications transactions in hardware, thus requiring expensive and bulky equipment to accomplish these applications.

All of the above-referenced patents disclose voice-based systems which are discrete and which perform narrow, limited voice-based transactions. If a customer needs a voice messaging system, a device such as that disclosed, for example, in the Matthews et al. patent could be purchased. However, if the customer also needs a system to interact with callers and to answer routine questions about the status of, for example, their bank accounts, a separate IVR system would be necessary. Similarly, if a customer needs to perform retail ordering and accounts management, a separate CP system such as that disclosed in the Mehta et al. patent must be purchased. Thus, it can be seen that the problem facing a customer who requires multiple voice processing functions is that of the proliferation of a multitude of special purpose systems that are expensive to purchase and to maintain, and which potentially process telephone calls in separate and disjoint manners.

An illustrative example will provide to those with skill in the art an appreciation of the magnitude of this problem. Consider a bank that allows its users to inquire about the balance of their accounts using an IVR system, but must now transfer a call to a VM system if the caller wishes to leave a message for an officer of the bank that could not be reached. This creates several problems for both the bank and the user. First, the bank must purchase and maintain at least two voice processing systems, an IVR system and a VM system. Second, the user must wait while one system addresses the other system to provide the new voice processing function. Third, the bank has no way of getting a consolidated report of the handling of a given call from start to finish. Fourth, if the user decides that since the bank officer is not available and the IVR system can provide additional information to answer a particular question, the transfer back to the IVR takes a considerable amount of time and is complicated since the user must usually enter the entire identification password information again, thereby leaving the bank without any way to trace a particular call as it is routed from one discrete voice processing system to the other.

As more discrete voice processing systems proliferate in a single environment, the problem of multiple disjoint systems becomes even more complex. There is a longfelt need in the art for methods and systems which integrate the various disparate voice processing functions to provide a voice processing system which effectively and economically provides all of the desired voice processing functions for a customer. This need has not been fulfilled by any of the prior voice processing systems heretofore discussed, which only focus

narrowly on one particular sub-technology in the complex and ever-growing array of voice processing sub-technologies.

A proposed solution to solve this long-felt need has been to connect a VM and IVR system together through a signalling link that coordinates the two systems. This link allows the systems to exchange calls with proper information relating to each call and which generates consolidated reports. However, the customer must still purchase discrete systems, and this solution is akin to suggesting that the customer purchase a personal computer with a word processing package of choice, another personal computer with, for example, a spreadsheet program, and yet another personal computer with a graphics program. Clearly, this is a cost prohibitive and ineffective way of performing a plurality of voice processing tasks and is not acceptable in light of the realities of today's business markets.

Another proposed solution to the integration problem has been to package two or more discrete systems in a larger cabinet. Usually, systems having a large cabinet have nothing in common except the cabinet itself. The systems may have their own separate consoles and keyboards, or they may have an A/B switch to share a single console yet still retain their individual keyboards. In all such "cabinet" systems, there is coexistence of applications but not integration of applications. Furthermore, systems which provide coexistence of applications usually provide hard-coded software in C-language, while the rest of the application development environment consists of C-language functions and programmer documentation that can only be understood by an expert programmer, but not by a customer who may require versatility and ease of use. Thus, the aforementioned integration attempts do not solve the long-felt need in the art for a truly integrated voice processing system.

Yet another attempted solution to the integration challenge has been to use a fixed VM system or a fixed IVR system and modify the resultant composite system to provide VM and IVR functions for execution in tandem on a common computer. The results of such machinations have been mixed, and the customer generally ends up with an inflexible VM or IVR system wherein the limitations and problems of one half of the system dictate the abilities and utility of the other half.

Examples of attempts at integration can be found in U.S. Pat. No. 4,792,968 to Katz. The Katz patent discloses a system of analysis selection and data processing for operation and cooperation with a public communication facility, for example a telephone system. See Katz, col. 1, lines 57-60. The systems disclosed in Katz provide methods of selecting digital data to develop records for further processing and allowing a caller to interface directly with an operator. See Katz, col. 1, lines 62-68. Another example of an attempt at integration may be found in U.S. Pat. No. 4,748,656 to Gibbs et al. The Gibbs et al. patent discloses an interface arrangement implemented on a personal computer to provide business communication services. See Gibbs et al., col. 2, lines 8-12. The personal computer interprets appropriate control signals which are then forwarded under control resident software to activate a telephone station set and provide communication services. See Gibbs et al., col. 2, lines 18-28.

Another integration attempt is disclosed in U.S. Pat. No. 4,893,335 to Fuller et al., which teaches a telephone control system that produces control signals which are programmable to provide a variety of control functions to a remote user, including for example, conferencing and transferring functions. See Fuller et al., col. 2, lines 7-44. However in all of the above-referenced attempts at

integration, only limited applications are achievable and significant problems of interfacing the different voice transactions are encountered. These aforementioned attempts at integration simply do not provide high level and effective voice transactions.

The inventor of the subject matter herein claimed and disclosed has also recognized another problem facing the task of integrating VM with IVR and other voice processing systems. Caller interfaces present a significant problem in integration since VM systems generally have fixed, hard-coded interfaces. In an integrated environment, this restricts the versatility of the entire integrated system, since it confines the system to the limitations of the original design of the VM interface. For example, if an IVR system provides voice responses to an airline for crew scheduling, it is unlikely that a IVR system could understand an employee number, translate it to an extension, look up the caller's supervisor and automatically transfer or drop the message in the supervisor's mailbox without querying the caller. The VM interface is usually inadequate to perform such complex tasking in an economical fashion. Thus, a fixed VM system quickly dominates the more flexible IVR system when the two systems attempt to operate together and the necessary VM caller interface is introduced in a pseudo-integrated environment. Such pseudo-integration schemes to put different voice processing applications together have heretofore simply not been able to accomplish the multifarious complex voice transactions required. Prior integrated systems do not solve the long-felt need in the art for a truly universal integrated voice processing telecommunications system.

During the evolution of the voice processing industry, VM systems have not been customized to perform according to a particular customer's unique specifications. Thus, VM-type systems were developed in mostly hard-coded traditional programming languages such as the C-language or Pascal language. In contrast, IVR systems were generally more sophisticated and employed primitive customization for particular applications. The IVR systems were thus generally designed in higher level programming language known as "scripted languages." Scripted languages merely replace the C-language or Pascal knowledge requirements of the system developer with that of the Basic language.

The common problem which emerges with the use of scripted languages is a disorientation of the system developer when designing the flow of the particular application. Furthermore, most scripted languages require several dozens of pages of basic code to accomplish even a simple programming task. Even though scripted programs can be interpreted by a programmer having less expertise than that which would be required if the software programs were written in the more traditional C-language or Pascal language, it will be recognized by those with skill in the art that after even a few pages of the lengthy scripted code have been reviewed, the entire flow of the application becomes disjoint and escapes the normal comprehension of even the most expert programmers in scripted languages.

In order to devise ways of alleviating the problems extant in scripted software voice processing systems, the concept of a state, event and action to define applications having programming methodologies in, for example, C-language or Pascal have been developed. Example of such systems are disclosed in U.S. Pat. No. 4,747,127 to Hansen et al. The Hansen et al. patent describes methods of implementing states, events, and actions to control a real-time telecommunications switching system. The methods of performing voice processing transactions in the Hansen et al. patent are accomplished using a scripted

base language similar to the "SHELL" programming language used by the AT&T UNIX System V operating system. See Hansen et al., col. 7, lines 15-35.

The methods and systems described in the Hansen et al. patent are strictly limited to telecommunications switches on a PBX. While the implementation of states, events and actions to perform higher level voice transactions is desirable, the systems and methods disclosed in the Hansen et al. patent do not fulfill the long-felt need in the art for integrated voice processing systems adaptable to provide multiple functions in a single, general-purpose computer environment and for varying customized applications. Furthermore, the use of non-traditional script base high-order programming language severely limits the adaptability of systems taught in the Hansen et al. patent, and thus the systems and methods disclosed in the Hansen et al. patent cannot be manipulated to provide integrated voice processing transactions.

The aforementioned plethora of voice processing systems are generally restricted to discrete sub-technology areas and accomplish narrow tasks of specific voice transaction functions. The above systems are at best only partially adapted to be customized for particular user applications and are not practically integrated to provide multiple voice processing transactions in a common computer platform. The aforementioned long-felt needs in the art have therefore not been fulfilled by any of the voice processing systems and solutions which have been developed, and do not provide integration, adaptability, customization, and flexible architecture definition. The present invention solves these problems and fulfills these long-felt needs.

SUMMARY OF THE INVENTION

Satisfaction of the above-referenced long-felt needs in the art is accomplished by the present invention, which provides an application development environment that allows customization of integrated voice processing systems. Systems and methods provided in accordance with the present invention further allow voice processing application developers and engineers to quickly develop and layer multiple voice processing applications together as customer needs change, without confining these developers and engineers to the narrow and focused specifications of prior voice processing systems. Whether a VM, IVR, CP, FAX, or other type of voice processing system is desired by a customer, customized systems provided in accordance with the present invention provide a standard operating system realizable in a general-purpose computer common-platform, environment readily available in the art and easily recognizable to system users. Furthermore, utilization of commonplace operating systems and standard interfaces between databases and the general-purpose computer, common-platform provide the systems disclosed and claimed herein with versatility, flexibility and the ability to provide integrated voice processing transactions. Thus, systems provided in accordance with the present invention are easy to use since they are customized to the customer's unique specifications, and are economical since they reduce the need for specialized and redundant computer hardware.

In preferred embodiments, the voice processing system comprises a general-purpose digital computer, common-platform adapted to communicate through the telecommunications lines with an outside environment; a relational database interfaced to the general-purpose, digital computer for storing at least one object having a state that is modifiable by a vector protocol, thereby producing a voice transaction event that is output to the user through the telecommunications line, whereby the user activates the vector protocol

through the telecommunications line to act on the object to produce the voice transaction event; and an interface adapted to convert user input commands to the voice processing system through the telecommunications line to activate the vector protocol and alter the object's state.

Methods provided in accordance with the present invention also solve a long-felt need in the art for adaptable, integrated voice processing systems which may be customized for a particular user's application or specification. With systems and methods provided in accordance with the present invention, any type of voice processing function commonly available today, or which will be necessary in the foreseeable future, can be integrated in a vectored-state architecture written in traditional C-code or other desirable programming language. Thus, methods and systems provided in accordance with the present invention allow for easy customization and adaptability of voice processing systems to provide a multitude of voice processing functions, and for trouble-shooting or modification of such systems.

Methods of performing voice processing functions in a voice processing system utilizing a digital input device and a telecommunications line are further provided in preferred embodiments. The methods preferably comprise the steps of receiving input from a user over the telecommunications line and storing the input in a digital format on a general-purpose computer-based control system, initializing a state vector function stored in a memory in the general-purpose computer with the digital input received from the user, operating the state vector function on a system-defined object having a state associated therewith stored on a database further associated with the general-purpose computer, thereby modifying the object's state, generating user-recognizable events which are realized by the state vector's operation on the object stored on the database, and outputting the user-recognizable events generated by the state vector's operation on the object in response to the change in the object's state.

The advantageous and beneficial results provided with systems and methods provided in accordance with the present invention satisfy long-felt needs in the art for integrated systems implemented on general-purpose computer common-platforms using standard operating systems and industry-recognized software interfaces. The unexpected and advantageous results achieved from creating customized voice processing systems which are adaptable to new voice processing applications, when confronted with particular customer demands, are extraordinary in light of the present state of the art. With prior voice processing systems occupying niche sub-technologies, the ability to accomplish new customer specified application demands is so limited as to be nearly nonexistent. Such demands will be increasingly made on the voice processing systems of the future because of the increased awareness of, and desire for, voice processing functions by business and other enterprises. Systems and methods provided in accordance with the present invention are designed to meet such demands.

DRWDESC:
BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of an integrated voice processing system provided in accordance with the present invention.

FIG. 2A is a block diagram of the interaction between vectors, objects, and events provided in accordance with the present invention.

FIG. 2B is a flow diagram of an exemplary voice processing system integrating call processing, voice messaging, and interactive voice response.

FIG. 3 is a block diagram of major processing elements found in voice processing systems in accordance with the present invention.

FIG. 4 is a state flow diagram of an exemplary play vector.

FIG. 5A is a block diagram of application logic state tables provided in accordance with the present invention.

FIG. 5B is a block diagram of a compiled next-vector stack element.

FIG. 5C, is a block diagram of a compiled speech stack element.

FIG. 6 is a state flow diagram of a voice window play vector provided in accordance with the present invention.

FIG. 7 is a state flow diagram of an exemplary voice window input vector provided in accordance with the present invention.

FIG. 8 is a block diagram of an exemplary voice window schema for a voice window vector.

FIG. 9A is a block diagram of an exemplary voice window field types for the voice window schema of FIG. 8.

FIG. 9B is a block diagram of exemplary voice window enunciation types for the voice window schema of FIG. 8.

FIG. 9C is a block diagram of available exemplary voice window field attributes for the voice window attributes for the window schema of FIG. 8.

FIG. 10 is a block diagram of an exemplary compiled vector element for the voice window schema of FIG. 8.

FIG. 11 is an exemplary compiled application logic state table for a voice processing system provided in accordance with the present invention.

FIG. 12 is a block diagram of a preferred method for creating dynamic SQL statements for voice windows.

FIG. 13A is a block diagram of an exemplary message envelope which integrates voice processing functions.

FIG. 13B is a block diagram of exemplary message segment attributes for the message envelope of FIG. 13A.

DETDSC:

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

Referring now to the drawings wherein like reference numerals refer to like elements, a telecommunications system provided in accordance with the present invention is illustrated. In preferred embodiments, the telecommunications system of FIG. 1 comprises a general-purpose computer common-platform which economically and efficiently integrates all of the sub-technology voice processing functions currently available, and others which it is contemplated will be developed in the future. In operation of the telecommunications system of FIG. 1, a user shown generally at 10 accesses a TOUCHTONE telephone 20, in preferred embodiments, to call a particular business or organization which utilizes telecommunications systems provided in accordance with the present invention. TOUCHTONE telephone 20 preferably digitally communicates over a telecommunication line shown schematically at 30 with a general-purpose digital computer 40 that provides the control and processing functions for the telecommunications system. The general-purpose, digital computer 40 is preferably adapted to communicate via telecommunications line 30 with the user 10 in the outside environment. The outside environment may generally comprise a number of users who are attempting to access the telecommunications system simultaneously and receive information from it.

The general-purpose computer 40 preferably provides an interface, shown graphically at 50, which is adapted to convert commands from the user that are input through the telecommunications lines 30 into a recognizable "query" to the telecommunications system. The interface 50 allows recognition of the user commands which have been digitized, so that the particular software routines which run the voice processing transactions can be recognized by the telecommunications system.

The interface 50 preferably communicates with a relational database means 60 which further communicates through the interface to the general-purpose computer 40 and which stores at least one object which has a state that is modifiable by a vector protocol. When the vector protocol operates on the object, it produces a voice transaction "event" that can be output to the user 10 over telecommunications line 70. The telecommunications line 70 and telecommunications line 30 may in fact be one integral line, or can be separate lines as shown here. The user preferably activates the vector protocol, which is also stored on the relational database 60, through telecommunication line 30 to act on the object and produce the voice transaction event. Interface 50 translates the user command into a standard query which is recognizable by the operating system of the general-purpose computer 40 as it communicates with the relational database 60.

The relational database 60 stores the vectors, objects and events which are used to drive the telecommunications systems provided in accordance with the present invention. It will be recognized by those with skill in the art that relational database 60 may in fact be an integral part of computer 40, or may be stored in a separate digital computer that is connected through yet another communications line to computer 40. In this fashion, many digital computers 40 may actually have access to a single relational database 60 should the particular customer application require such an arrangement. Whether the relational database 60 is an integral part of general-purpose computer 40 or a separate database at a remote location, all such arrangements and equivalents thereof are contemplated to be within the scope of the present invention.

Furthermore, interface 50 may also communicate user queries to, for example, other mainframe computers 80, or yet other outside computer based hardware 90

that is adapted to understand the queries posed through interface 50 and further provides voice transaction events through telecommunications lines 70 after an object has had its state modified by a vector provided in accordance with the present invention.

In yet further preferred embodiments of telecommunications systems herein described, the user receives voice transaction events at a handset 100 which is actually part of the TOUCHTONE telephone 20. The voice transaction events which are output to the user, shown generally at 110 may, for example, prompt the user to activate yet another vector protocol in a telecommunications system stored on the database 60 or other computer equipment, or prompt the user to end his or her communications with the system.

One of the main features and advantages of the present invention resides in the fact that the application development environment and telecommunication system itself is realized in a general-purpose computer common-platform with industry standard architecture. This includes an industry standard operating system, industry standard database, and industry standard communications protocols. In further preferred embodiments, the general-purpose computer is an American Telephone & Telegraph (AT&T) 386-based personal computer utilizing the UNIX operating system. In still further preferred embodiments, interface 50 comprises the "standard query language" (SQL) interface which is an International Business Machine (IBM) standardized interface that is recognizable by a large number of databases and computer systems commonly available in the industry today.

The SQL interface converts the user's commands input over the telecommunication line to standard queries which can be communicated to the relational database 60, mainframe computer 80, or other computer hardware equipment 90 having vectors, objects and events stored thereon. In still further preferred embodiments, the relational database 60 provided in accordance with the present invention is also an industry standard database which communicates with general-purpose computer 40 through the SQL interface. The inventor of the subject matter herein claimed and disclosed has determined that the IBM SQL standard based INFORMIX database is a preferable database which can be accessed by the user and which stores the vector protocols, objects, and events used by the voice processing systems provided in accordance with the present invention. While other standard databases and interfaces may be utilized in voice processing systems having state vector architecture as described herein, the industry standard SQL interface and INFORMIX database have been determined to provide storage and acquisition systems which are widely used by all facets of the computer industry. Since these standard architecture systems are preferably utilized in accordance with the present invention, telecommunications systems described herein are easily modifiable and adaptable for virtually any customer application or to any particular specification of voice processing transactions known today or which will become known and desired by customers in the future.

Through the use of the SQL interface with standard TOUCHTONE telephones, telecommunications systems provided in accordance with the present invention interface to the INFORMIX database and various other computer hardware for sophisticated applications which are required by certain customers and which cannot be integrally provided by any other voice processing systems in the present sub-technology voice processing groups. Telecommunications systems described herein are thus completely configurable to any telecommunications network, and can be adapted to perform all of the various functions presently

provided by the separate sub-technology groups. These advantageous results have not heretofore been achieved in the voice processing art and represent a significant long-term solution to the integration of voice processing functions which until now have functioned in discrete operating environments, thereby greatly increasing the costs to customers of operating with one or more of these different and discrete voice processing systems.

In accordance with the present invention, customer applications of telecommunications systems are developed by using programmed "objects," determining the state of the programmed object, performing an action on the object, determining a computer event caused by the application of the action onto the object, thereby producing a voice transaction, and disposing of the event so that the process may be repeated if so desired. The disposition of the event may cause another action on the same object or, in preferred embodiments, it may force a permanent or temporary change of reference to one or more objects.

Objects provided in accordance with the present invention are defined on a microscopic scale to perform single functions and for detailed operations. However, groups of objects provided herein by the integrated voice processing system of this invention may also be formed to facilitate operations on a higher level. To this end, objects may be as detailed as, for example, a spoken name contained in a VM application of a person sending a voice message, or as broad as an entire mailbox containing all voice messages, facsimiles, or electronic mail for an individual.

Similarly, actions may also be microscopically defined for a high degree of detail, while at the same time they may be grouped together in order to hide unnecessary detail from the user accessing the telecommunications system. A detailed action may, for example, play a spoken name from within a VM application, and another action may play, for example, the date and time that the message was sent, while yet another action plays only a particular desired segment of the message. Detailed actions may also be combined into a group of actions which are accessed on the relational database by a single reference commanding a voice message envelope to play the entire contents in a manner determined by arrangement of the detailed actions within the group. All such applications of objects, actions and events are encompassed within the scope of the present invention.

In still further preferred embodiments, multiple objects, actions, events, and event disposition instructions may be combined together to form an "Application State Logic Table" 11 (AST). An AST implements all functions and features of a VM system and comprises detailed individual objects and actions as well as potentially several smaller groups of objects and actions. However in accordance with the present invention, the AST may be treated as another single object, thereby allowing flexibility of integrating multiple applications.

The actions, events, and objects described herein are adaptable and designed to process any application in the voice processing universe. In accordance with the present invention, processing a voice transaction is facilitated through the use of "vector" protocols. In reality, vectors are software programs which perform the various functions that a customer desires for a particular voice processing system. Since each telecommunication system described herein is an object-oriented, event-based system adaptable to perform any of the functions presently available in the voice processing rubric, each system is "vector

customized" depending upon the particular voice processing features which a customer may desire for its system. Preferably, the vectors are programmed in C-language but application developers may merely use them by reference and need not concern themselves with the details of the C-language code, thereby allowing application developers not fluent in scripted computer languages to develop applications quickly and to provide layered multiple applications as the particular needs change without having to be confined by the specifications of an earlier application.

Since the vectors are not "hard-wired" and are adaptable through changes in the C-code, telecommunications systems provided in accordance with the present invention completely encompass all types of voice processing applications. Referring to FIG. 2A, an illustration of the interaction of the vectors, objects, and events is illustrated. The vectors 120 are stored on the computer in the INFORMIX database. In order to facilitate speed and ease of operation, the AST comprising a plurality of vector references and arrangements is compiled rather than used in source code form. Many different kinds of vectors are available and can be accessed by the user. The vectors 120 thus generally comprise a number of routines such as, for example, "play a message" 130, "record a message" 140, "dual tone multifrequency input" (DTMF) 150, "call back" 160, "transfer" (XFER) 170, and any other vectors which are particularly desired to implement a particular customer demanded voice processing function.

The above-referenced vector protocols provide many different kinds of voice transactions which heretofore have only been available in separate discrete voice processing systems from the different sub-technology areas. For example, the "play a message" vector 130 and "record a message" vector 140 are generally recognizable as VM functions. However, the send host message vector 150 and the wait host response vector 180 are generally recognizable as providing IVR-type functions. Similarly, the XFER vector 160 is, for example, recognized as a CP function. Thus, by using "soft-wired," that is, software programmable vectors which are easily modifiable and adaptable in the C-language, many different kinds of voice processing functions are available on a single telecommunications system provided in accordance with the present invention. Thus, a customer need not purchase many different types of voice processing systems, and the resulting customized vector-oriented systems described herein are generally two to two-and-a-half times less expensive than hybrid, pseudo-integrated voice processing systems which have been used before. This presents significant cost savings to the customer and evinces the great advantages inherent with the telecommunication systems provided in accordance with the present invention.

In further preferred embodiments, the vectors 120 operate on a number of system-defined objects 190. The objects 190 are preferably also stored on the INFORMIX database which may be an integral part of the general-purpose computer or may be found at a location remote from the computer. The objects may also be stored in other mainframe computers or computer hardware devices when a particular object must perform a function associated with these other remote systems. The objects have "states" associated therewith which correspond to a particular place that the object is in time, after having been operated on by a vector protocol, or which may be an existing initial state before such operation. Depending upon the extent and sophistication of the telecommunications system wherein a number of users might have simultaneous access to the system causing vectors to operate on the objects, a multitude of states may exist for the object at any one time in light of the voice transaction event produced by the object that will be output to the particular

users.

The vectors 120 operate on the objects 190 to change the objects' states in response to users' queries to the system. In further preferred embodiments, the telecommunications system is adapted to keep track of all of the state operations performed on the objects in time so that the particular states of each of the objects are always available to the system. The vectors 120 change the state of the objects by performing actions 200 on the objects defined by the vectors. Vectors may ask, for example, to call a digitized voice for a VM function, an account password for an IVR voice transaction, or to call a number of software attendants in a CP system. Each time the vectors 120 act on the objects 190, the objects, states, are changed, thereby producing events 210. The events 210 are output to the user in a recognizable fashion so that the user can make a decision concerning which way to proceed as it accesses the telecommunications system.

Many types of voice processing transactions or events are available to provide voice processing functions. For example, a "time-out" (T/O) event 220 holds the system in a particular mode for a predetermined amount of time. Additionally, "error" events 230 and "OK" events 240 are available to the system, events which require additional user "input" 250 may be implemented in the system, as well as other events which provide sophistication and more involved user interaction with the system. The events preferably produce user-recognizable output, or promptings to the vector protocols in an AST environment, to produce new object states and further events which may finally be output to the user. Any voice processing functions which have been implemented in the past on discrete systems are thus programmable in a telecommunications system provided in accordance with the present, and the events so produced are stored on a database to be available to a user when the user accesses a particular system.

Referring to FIG. 2B, an exemplary flow diagram of state vector operation in a telecommunications systems provided in accordance with the present invention is shown. The particular exemplary embodiment shown in FIG. 2B integrates a number of voice processing functions, wherein circles represent vectors, rectangles represent objects, and triangles represent particular events output by the system.

As the user accesses the system, a "welcome" object 260 is first reached which provides a welcoming recording heard by the user when the system picks up. A "play" vector 270 acts on the welcome object to change its state, thereby producing an "OK" event 280 which outputs the voice transaction found in the welcome message. However, if the user accidentally inputs a digital signal from the TOUCHTONE telephone, thereby forcing the play vector 270 to operate on the TOUCHTONE telephone object 290 for example, the TOUCHTONE telephone object in a new state produces an "error" event 300 which is output to the user. In such a situation, a "return" vector 310 is then accessed on the relational database to operate on the welcome object, thereby producing yet another OK event 320 and replaying the welcome message, this time perhaps with a warning to wait for a next output prompt.

In this example, the OK event 320 causes play vector 270 to change the state of the welcome object 260 which then further outputs another OK event 330. At this point, another vector 340 plays a message to the user. After the user hears the particular voice transaction message, the system then waits to determine

whether a T/O event 350 occurs, that is, the user has not taken any action to access another vector. If T/O event 350 occurs, then yet another vector is accessed 360 which acts on the operator object "0" at 370, thereby connecting the user with a live operator to determine if there is an input problem or misunderstanding. Other events, vectors and objects could then be accessed further by the user at 380.

However, if the user does not allow a T/O event to occur and wishes to record a message in the system, a record vector 390 is accessed, producing an event, "1" at 400 indicating to the system that a message will be recorded by the user. The system then accesses an "end" vector 410 to output an ending signal to the user and to instruct the user to hang up the handset. Alternatively, other vectors are accessible by the user after recording a message, the other vectors 420 producing yet other event-based voice transactions which are recognizable to the user. The only voice processing limitations placed on the system of FIG. 2B are those found in the specifications for the particular system required by the customer.

Upon examining the exemplary voice processing system illustrated in FIG. 2B, it can be seen that even this simple system integrates many aspects of voice processing functions which were only traditionally available on discrete systems. Since play vectors 270 are available, the system behaves as a VM system. Since the basic functions required to implement VM are not unlike the functions required to implement IVR and CP, the same vectors are used for multiple purposes in preferred embodiments. On the other hand, since messages are played to the user according to the play vector 340 and may be recorded by the user according to the record vector 390, the system also integrates basic VM functions. Similarly, since the user can access event-based attendants or operators at 370 according to the call vector 360, CP aspects of voice processing transactions are also available. The customer who has provided the specifications for the system of FIG. 2B has therefore attained a very versatile customized system which has eliminated the need for three discrete VM, CP, and IVR voice processing systems that would be available from separate vendors. The state vector architecture operating on system-defined events thus provides flexibility, and allows for total integration of what have previously been completely separate, unrelated voice processing tasks.

Referring now to FIG. 3, a more detailed block diagram of the main processing elements of integrated voice processing systems provided in accordance with a preferred embodiment of the present invention is shown. Preferably, communication logic state tables 430 are generated by the system so that processes found and generated by data communication servers 440 can perform run-time operating instructions for the system. The communication servers 440 are responsible for communication with external computers and other systems which interface with voice processing systems described herein. Yet another server 450, which is an alarm/log server, is interfaced with the data communication servers 440 to provide alarm and other type functions for the system.

The heart of the voice processing system is the runtime executive (RTX) block 460. RTX 460 in a preferred embodiment provides the main control processes for voice processing systems provided in accordance with the present invention during operation. RTX 460 is an object-oriented, run-time programmable finite state machine. The system alarm/log server 450 handles messages from all other servers in the system and logs and stores the messages. Additionally,

alarm/log server 450 controls a triggering of system alarms, including audible, visible, and hard contact closures for remote sensors. In further preferred embodiments alarm/log 450 also manages the logging of system activity and usage data provided by the RTX processes. The alarms are stored in an alarm log 500 which is a data table.

The RTX preferably executes instructions which are stored and contained in an application logic state tables (AST) block 470. The application logic state tables found in AST 470 in preferred embodiments are generated by an application editor 480 (APE) which is preferably a 4GL application editor. Furthermore, application logic state tables are also generated by a GUI application editor (APEX) shown at block 490.

The APE 480 allows application developers to arrange vectors provided in accordance with the present invention to formulate the applications desired by the customer. Similarly, APEX 490 is preferably a formatted, screen interactive development program that uses a graphical interface instead of using line graphics and text-based interfaces. APE 480 and APEX 490 thus generate the communication logic state tables stored in block 430. The state tables for voice processing systems provided herein which are stored in block 430 and AST block 470 are further coordinated by APE 480 and APEX 490 to facilitate asynchronous processing of the application logic.

In a further preferred embodiment, the offloading of communications to the data communication servers 440 allows RTX 460 to issue requests and receive responses without blocking data flow. Additionally, because data communication servers 440 and RTX 460 are interactive, it is not necessary for RTX 460 to be concerned with the particular details of protocols and external computer interfaces to the system. This provides the advantageous result that the processing of the voice transactions is divided into two sets of processes, thereby resulting in inherent support for network-based processing.

Preferably also, the data communications servers 440 are driven by object-oriented application logic found in RTX 460. The SQL servers 510 function very similarly to data communications servers 440 and are responsible for front-ending relational databases, and enabling the RTX 460 to interact with relational databases using SQL statements. Thus, RTX 460 dynamically generates SQL statements during runtime to store and retrieve information to and from relational database 530 which can be considered as the system database.

The SQL processor 510 executes the SQL statements on behalf of the RTX 460 which allows RTX 460 and SQL server 510 to operate asynchronously. This further provides the advantageous operating result of a division of processing between the two separate sets of processes which promotes network-based processing in a true front end/back end, client/server processing environment.

The exchange of information between RTX 460 and SQL server 510 or alternatively between RTX 460 and data communication server 440 is preferably conducted in welldefined boundaries across individual domains. This localizes processing and reduces the flow of unnecessary data which is encountered in prior storage servers of other network-based operating systems. An outcall server 540 is responsible for generating "outcall lists." Outcall server 540 is driven by SQL statements and selects telephone numbers for outcalls. The SQL statements for the outcall server 540 are preferably entered by a human operator or alternatively, they may be generated by the RTX processor 460. In further

preferred embodiments, the output of the outcall server 540 is channelled to the RTX 460 which is then responsible for placing the outcalls. The outcall server 540 may supply only limited information about each outcall to RTX 460, or it may supply detailed information. In either case, the RTX 460 may request further information from the same database that outcall server 540 used to make the selection in the first place by communicating through the SQL server 510.

The exchange of information between RTX 460 and the various servers available to the system is preferably organized by the RTX in internal storage using a concept called "voice windows." Information is exchanged between the voice windows stored on RTX 460 and its associated servers using formatted buffer exchanges over UNIX communications facilities.

A broadcast server 550 provides mail sorting and distribution for the integrated voice process systems provided in accordance with the present invention. Broadcast server 550 again preferably uses SQL statements to retrieve information from the relational database 530, thereby making sorting and distributing decisions. In preferred embodiments, broadcast server 550 sorts, collates, deposits, and broadcasts mail, including voice messages 560, facsimiles 570, and electronic mail 580, as well as other voice processing applications which are desired by the customer by its particular voice processing system as described herein. Broadcast server 550 also provides network transmits and distribution of inbound network mail. The broadcast server thus periodically cycles through pending requests and periodically looks for new requests. In addition, RTX 460 may invoke broadcast server 550 for "time special" or urgent deliveries.

RTX 460 is a finite state machine and it is dynamically programmed at run-time with instructions from AST 470. These application logic state tables are generated by either APE 480 or APEX 490 which allow application developers to specify the placement of objects, actions to be taken on the objects, and the handling of various events that result from the specified action.

Referring now to FIG. 4, there is illustrated an exemplary vector which encompasses an object (in preferred embodiments a speech phrase), object states, an action, and a disposition of various events. The vector may have several states, events, and disposition of events internal to it; however, there is in general provided only one entry and exit point for the vector. The vector enters at 610 where the circles in FIG. 4 represent events such as "wait digits" 620, validation 630, T/O limit 640, and "done" where the vector exits at 650. The action of this particular vector is to play a speech phrase. Various disposition events are available such as an OK event 660, disconnect event 670, a "play done" event 680, a "digit input received" event 690, a T/O exceeded event 700, a disconnect event 710, another "error exceeded event" 720, a digit validation event 730, and yet another OK event 740. Thus, the play vector begins at 610 with an entry played so that the system waits for digits at 620, checks for digit validation at 630, or a T/O limit at 640, potentially checks for error limits at 600 or T/O limits at 640, and exits at 650 following a disconnect 670. Upon exiting at 650, the vector calling mechanism checks the exit status of the vector and consults a vector's "next-vector stack" for the dispatching of the next vector.

State Table Architecture

With reference to FIGS. 5A, 5B, and 5C, the relationship of the various logic state tables that constitute the vector of FIG. 4 are shown. In the vector

table of FIG. 5A, a vector 750 is the current vector that has been dispatched by the voice processing system. Associated with the vector table is an extended vector table having a vector 760 which provides "attributes" for the vector 750. The vector's attributes are the particular codes which direct the vector to perform state modification of an object as described herein. In the next-vector stack table, a next-vector stack 770 is accessed by the vector 750 to point to the next-vector available in the stack. Similarly in the speech stack, a next speech stack 780 is accessed by vector 750.

Vector 750 accesses an extended vector 760 through a reference 790 which references through the top of the extended vector stack. Similarly in the next-vector stack, the vector 750 references the next vector 770 through a reference 800 to the top of the next-vector stack. Speech associated with the vector is referenced through 810 on the speech stack. Reference 810 references through the top of the speech stack elements.

SQL statements associated with the vector are referenced through a reference 830 into an extended parameter stack table to a specified entry 820 within the table. The vector 750 comprises codes which tell the system, for example, which extended vector to use, which extended parameter is next, which vector is next in the stack, and which speech element is next in the stack. A plurality of vectors so described having the elements 750 through 830 are stored in the AST 470 as a set of logic state tables.

FIG. 5B shows the compiled next-vector stack element in an exemplary embodiment. The next-vector stack element may comprise a "key" field 840 which points the vector to the next event available in the system. The next event shown in this exemplary embodiment is a "macro call" in field 850 which tells the vector to call and search for the next vector stored in field 860. In a similar fashion, a compiled speech stack element shown in FIG. 5C comprises for example, an archive field 870 which tells the vector where to go in a database to retrieve a speech element, according to a particular format in a format field 880. An offset field 890 holds a code to tell the vector at what point in the archive to retrieve the speech, and a length field 900 tells the vector the particular length of the particular speech element which will be retrieved.

Voice Window Architecture

As discussed above, in preferred embodiments the exchange of information between RTX 460 and the data communications server 440, the SQL server 510, and outcall server 540 is organized using voice windows provided in accordance with the present invention. Referring to FIG. 6, there is shown an exemplary voice window protocol organized to manage the information exchange between the processes heretofore mentioned, in between the particular voice transactions and uses of the system. In the exemplary embodiment of FIG. 6, the state transitions within the voice window vector for an output or protected field are illustrated. The voice window structure of elements plays data at 910 and begins with a play label 920 that acts as the entry point for the vector. Similarly to the play vector shown in FIG. 4, the vector shown in FIG. 6 has an object which in this case is the voice window field, has several states such as the "play data" state 910, "play a label" state 920, the "next field" state 930, and a "done" state 940, which is the exit. The vector of FIG. 6 plays the voice window field, and contains a disposition of various events, for example, event 950 which is "play done," event 960 which is "digit received," event 970 which is "field found," event 980 which is "disconnect," event 990 which is "done play data," event 1000 which is "digit," event 1010 which is another "disconnect," and event 1020

which is "no next field."

Upon entry at play label 920, the vector executes the first field in the voice window and continues execution through the entire set of voice window fields, unless prematurely interrupted by a terminating event. Using this vector, an application dispenses information to the caller one information field at a time. On entry at play label 920, the vector preferably plays the field label associated with the first field, and either plays to completion and leaves the play label state through event 950 play done, encounters a caller input event 960, or gets a disconnect signal from a telephone switch at disconnect event 980. If either done event 950 or digit event 960 are encountered, the state preferably transitions to state 910 play data, and the system commences playback of the data contained within the voice window field.

Play data state 910 is terminated by play done event 990, digits received event 1000, or disconnect event 1010. In the case when done event 990 or digit received event 1000 is encountered, the program then dispatches the next field state 930 which causes the program to bump the field pointer to the next voice window field available in the vector.

If next field state 930 is able to bump the field pointer, the cycle starts over with found event 970 and transitions to play label state 920, otherwise the vector is prepared for exit through the done event 1020 and the done state 940. In cases where the program is interacting to one of its states through the telecommunication channel, a disconnect signal from the telephone switch that is connected to the telephone channel will cause immediate exit of the vector through done state 940. Upon exit from this vector, the vector calling mechanism checks the exit status of the vector and dispatches the appropriate vector based upon the entries and the returning vector's particular specified nextvector stack. This is identical to the return to post exist processing of the play vector exit through 650 as discussed in reference to FIG. 4.

State Transitions

To better understand the voice window state transitions extant in FIG. 6, FIG. 7 illustrates the allowed voice window state transitions. When executing an input allowed field, the vector entry point at 1040 is a "play label" state. The vector steps through all the voice window fields, plays the voice field label through play label state 1040, and if a "digit input" is received from the caller at digit event 1130, the next state dispatched will be "get digit" state 1030, which gets the digits and waits for the number of digits expected. If all the digits are received in time, a "digit" event at 1160 causes dispatch to the next state to the "populate" state 1070 which populates the voice window field with the digits. The populated event 1070 then returns and the next field state 1080 which bumps the field pointer to the next voice window field after the done state is reached through a "none" event 1170 signifying that no next field is available. If "next field" state 1080 finds another field in the voice window, then the cycle starts over with the new field through "next field" event 1110 to the play label state 1040. However, if a T/O event 1150 takes place after the "get digit" state 1030 occurs, then time out event 1150 dispatches the "T/O limit" state 1050 which checks for the number of time outs that have taken place so far, and either causes an "OK" event 1090 to dispatch the play label state 1140, or causes the exceeded event 1040 to dispatch the done state 1060.

The combination of vectors in FIGS. 6 and 7 can be used to create a "super-vector" which handles both output or protected fields similar to the

play vector shown in FIG. 4, in preferred embodiments. This super-vector may then be deployed to handle the exchange of information between the system processes and voice transaction events to the users. Using the super-vector on voice window fields, the caller and the system programs are able to exchange information with a mixture of both protected and unprotected voice window fields.

Refinements and enhancements may be added to the vectors illustrated in FIGS. 4, 6 and 7. For example, information validation prior to population of the voice window field at population state 1070, verifying the data input to ensure that direct information is entered by the caller, reentry of a date, opportunity to accept partial input which is used when the program is able to narrow down from the partial input when a caller is attempting to offer appropriate help, and others are all potentially addable to the system. Additionally, voice programs may be used to build program/caller interfaces, which are interfaces used to exchange information between clients and the servers 440 and 510. For example, RTX 460 may request SQL server 510 to reply with the results of an operation and any information retrieved from the relational database 530. As part of this request from RTX 460 to SQL server 510, RTX 460 attaches a reference to one of its voice windows which receives the results of the database operation. When SQL 510 completes processing of the request, it responds back to RTX 460 using the reply to address contained in the original request. Upon receiving of a response, RTX 460 reloads the voice window reference and populates the window with information contained in the response packet. In this fashion, voice windows provided in accordance with the present invention are used to exchange information between users and the user interface program, and between user interface programs and relational databases.

Thus, the use of voice windows along with the creation of SQL statements from voice windows provides an extremely powerful and flexible method of facilitating user interfacing to relational database through SQL statements. The cooperation between RTX 460 and SQL 510 may also be extended in accordance with this invention, to cooperative processing between RTX 460 and the data communication servers 440. In this embodiment, data communication server 440 perform server functions by interacting with an external computer directly using SQL statements, wherein the external computer is in an environment which is foreign to the integrated voice processing system. For example, data communications server 440 may be a terminal emulation program using IBM 3270, Unisys Poll/Select, or UNISCOPE with Unisys terminal emulation, IBM 5250 terminal emulation, digital VT 100 terminal emulation, IBM LU6.2 peer-to-peer communications, or any other variety of communications protocols. Further in the case of the LU6.2 pair-to-pair protocol, the peer process on the foreign host external computer may also be an SQL server acting on an SQL front-ended database residing on the foreign machine.

Vector State Logic

While the data communication server 440 is custom designed for any type of communications protocol, the communication logic state tables 430 contain the vectors which define the "personality" of the data communication server in each customer application. Thus, as with the RTX 460 assuming multiple personalities based upon the arrangement of vectors in the application logic state tables 470, a single IBM 3270 server program in preferred embodiments serves multitudes of applications based upon the arrangements of the vectors in the logic state tables. By using vectors and windows with dynamic SQL statements, the flexibility inherent in creating powerful interfaces between the users and the

computers in a myriad of applications is multiplied. The interface so created is not limited merely to users communicating with computers, but could also be used to communicate between two computers in a voice processing system, wherein one of the computers emulates a user. The use of this kind of computer to computer interface is especially adaptable to customer specification requiring stress testing or benchmarking systems applications.

Voice Window State Logic

Voice windows can be viewed as collections of related voice fields, in a manner similar to a display in graphical window systems which are collection of display fields on display terminals. Each of the voice fields has particular characteristics and attributes, and an application developer can define a voice window in accordance with the present invention similar to defining the layout of a display screen in a display window system. Once a voice window is so defined, it is then referred to by vectors as objects to operate on. Each voice window will then have at least one voice field and a name indicating the ownership of the field by a specific voice window.

Referring to FIG. 8, a voice window schema is illustrated wherein each voice window has a name in a window name field 1190. The field name 1200 identifies the field within a window wherein window name 1190 and field name 1200 together represent a unique, fully qualified name for the voice window. This combination is used to reference the field throughout the application logic state tables 470. Label 1205 defines a field type in the voice window and label 1210 delineates the input size which preferably is a number of digits that are solicited for input from a caller. Label 1220 contains the voice label for the particular field which is much like the title or a heading for the information displayed on the display terminals in a window graphics system. Voice label 1220 is further preferably played by the vector window before either the data contents are played, or an input is solicited. The cross window name 1230 contains the name of the voice window that is used to resolve the window type by referencing to a cross window which is simply a window which works in tandem with the present voice window to accomplish a voice transaction. Cross field name 1240 contains the name of the field for resolution of the cross field reference as described above. Validation function 1250 contains the name of the optional user written, C-language function that may be called upon for data validation or transformation. Validation functions in preferred embodiments are bound at run-time and need not be bound together at link time to the RTX run-time executive.

A conversion format 1260 is used to perform transformation of data while populating the field in the case of unprotected fields and announces the field contents on protected fields. An enunciation type 1270 preferably defines the form of enunciation to be used for playing data from a field. Input delimiters 1280 provide a list of digits that will signal an end of the input from the caller whenever any one of the digits is encountered. Finally, field attributes 1290 modify the program behavior of the vector executing the field with a further combination of attributes. In the case of window field types 1205, window enunciation types 1270, and window field attributes 1290, various possible values for each of these elements are possible. Referring to FIG. 9A, voice window types such as a character 1300, a filler 1310, or a simple time stamp 1320 are possible window field types. In the case of a unique file name 1330, a particular data file type that causes the field to be populated by a string of characters representing a file name that is guaranteed to be unique across the entire system is specified. This particular file name may be used

to deposit various forms of messages for recipients. It may then be used without modification for broadcast to other recipients within the same system or the network without the possibility of file name collisions.

A "constant" field type 1350 identifies a constant within the voice window to be applied to the vector transformation. The voice window field type "reference" 1360 indicates that the field contains no data and that all attributes and characteristics are to be applied to data residing in another field either within the same voice window or across another window. The "copy" voice window field type 1370 preferably states that this field is to be populated with data from another field, either from within the same window or across another window. The "sequence" voice window field type 1380 causes the contents of this particular field to be incremented each time this window is executed.

Referring to FIG. 9B, the enunciation types available to this particular voice window scheme are illustrated. The enunciation type "date" 1390, "money" 1400, "numeric" 1410, and "paired numerical" 1420 provides these particular enunciations to the voice window and are common voice processing functions. Enunciation type phrase 1430 is used when certain data are necessary as a key to locate a prerecorded voice "phrase" to be played to the caller instead of the data itself. The enunciation type "time" 1440 is populated from a C-language formatted time function call to play the time to the caller. Finally, the enunciation types "percent" 1450 and "none" 1460 provide these particular selfexplanatory enunciations to the voice window for play to the caller.

Referring to FIG. 9C, attributes which modify the program behavior of the vector executing the field with a combination of attributes are shown. The "verify" attribute 1480 causes the field input to be solicited twice and compared before populating the field. "Partial input" attribute 1490 allows partial input of data to be accepted into the field, which facilitates the program to make an educated guess at what the caller is trying to accomplish if and when the caller is having difficulty using the system. The "continue on error" attribute 1500 disregards errors caused by the input or output of the field data with a resulting premature interruption or execution of the voice window by the executing vector. Finally, the field attribute "non-volatile" 1450 holds the system in a non-volatile state when certain input is received from the user requiring particular wait times.

The detailed structure of compiled vector elements as they exist on the application logic state tables is illustrated in FIG. 10. The vector name segment 1520 identifies the vector and the vector type segment 1530 defines the action that the vector performs on the objects that it operates on. A "sub-type" segment 1540 further defines the action that the vector performs when yet more complicated actions are required. A "parameter" segment 1550 contains information that is interpreted by the vector based upon the type 1530 and sub-type 1540. In preferred embodiments, whenever information in a parameter 1550 is used by a vector, that information becomes the object of the vector. The parameter 1550 contains any one of several parameters, including for example, voice window names, voice field names, phone numbers, dates, times, call durations and other particular parameters defined by the particular vector in use.

In preferred embodiments, any time a value is specified as a parameter to a vector in parameter 1550, the value itself may be substituted for by a voice field name which creates a dynamic object passing mechanism that is resolved

at run-time based upon information contained in the voice windows. This type of parameter information may come from the caller, from the relational database, from an external computer, or it may be a combination of all of these sources. Because of the flexibility of parameters 1550, the dynamic run-time modification of the execution logic in accordance with this invention is maximized.

An extended parameter segment 1560 contains the skeleton of the SQL statement which is used to create dynamic SQL statements using information from the voice windows. A next-vector stack segment 1580 preferably contains a plurality of next-vector dispatching instructions which are matched up against vector exit conditions. Similarly, the speech stack 1590 contains a plurality of speech references into the speech archives. These particular speech references are used to locate speech fragments to play to the caller when particular vectors call for the playback of speech on demand.

Referring to FIG. 11, an illustration of the structure of the application logic state tables is shown. A compiler version and time stamp 1600 ensures that the APE 480 and APEX 490 mark the state table so that a compatible version of RTX 460 is user to execute the application logic. A compiled channel descriptor 1610 contains information about the various telephone channels that will be serviced by the voice processing system. General information about the type of channel and its signalling characteristics are maintained in the compiled channel descriptor segment. Similarly, speech archive descriptor 1620 contains information about the particular speech segments that are used by the system.

An input/output (I/O) translation table 1630 holds an index for locating speech phrases based on data contained on voice window fields if called for by the phrase enunciation 1430, which is also used to translate input into window fields. Using input translation, caller inputs may be located in the index and substituted with the contents of a translation entry whose key matches the caller input. To avoid collisions with data in a voice window which is the same as data in another voice window but applied differently, the conversion format 1260 is applied to the data before performing look-up in the I/O translation 1630.

The plurality of vectors segment 1640 contains the compiled vectors, and the plurality of voice window segments 1650 preferably contains a plurality of voice windows. Voice windows thus appear in window fields consecutively stacked together from the first field to the last field. Each window is referenced by a window header which preferably contains information about the size and data area required to hold the field contents of the window. Windows are preferably read from a plurality of voice window segments 1650 dynamically created and destroyed in the main memory of the vector executing program at run-time. When the particular caller terminates the call, all the windows are destroyed in the main memory and the data area which has been reserved for the windows is preferably released. However, some window fields may be marked "non-volatile" by the field attribute 1510 and these windows are then used to maintain information between calls, thereby serving as a parameter passing mechanism between particular calls. This allows the system to behave in a dynamic fashion transferring data from caller to caller in preferred embodiments.

SQL Statement Creation

It is generally desired to create dynamic SQL statements from voice windows as described above. Referring to FIG. 12, a flow chart for transformation of SQL statements at run-time to executable statements from data in voice windows is

illustrated. A dynamic program searches for a token 1660 which are SQL components used a building blocks for SQL statements. If all the tokens have ended at 1680, there is a normal exit from the program at 1670. Otherwise, the token is written to the window field at 1690. If there is not a token in the window field at 1690, then the token is written at 1700 and the next token is accessed at 1660. Otherwise, at step 1720 it is determined whether the window is indeed active with a present token. If not, then the method defaults to 1710 with an error and the program is exited. If default does not occur, data is written at 1730 in a field and the method returns to 1660 for the fetching of the next token. In this manner, the SQL statements are transformed at run-time into executable statements in the voice window architecture.

Message Envelope in a VM System

With voice processing systems provided in accordance with the present invention, the main object of the manipulation of data by vectors is accomplished in integration of IVR, VM, facsimile, electronic mail (E-mail), CP, and other types of voice processing functions. A "message envelope" is created to provide integration as is illustrated in FIG. 13A. At block 1740, a "magic number" which identifies a particular file as a message envelope and the version of the program that created the envelope for backward compatibility is accessed. A segment count 1750 contains a count of message segments within the envelope, each envelope containing at least one segment. However, the envelope may contain several other segments of the same media type, or it may contain a mixture of voice, FAX, E-mail, or other data. Multiple segments of voice processing functions are used to keep a voice message intact while it is being forwarded from one mailbox to another with an adaptation from each mailbox owner. Additionally, multiple segments are used when a FAX or an E-mail message is sent or forwarded to another mailbox owner. In this fashion, multiple media message formats are contained within a single envelope.

A segment size block 1760 contains the size of the segment, while a sender identification (ID) block 1770 provides an identification of the sender of the message in the segment. If the sender ID is not known, this field will contain a zero for telephone calls from the outside where the caller ID is not available. The time sent block 1780 preferably contains the time when the message was created. Message attributes 1790 contain a combination of attributes for the various messages. Message format 1800 indicates the format of the message which is contained in a contents segment at block 1810. This format specification includes media and data compression indicators. The content segment 1810 is preferably used by the RTX 460 to determine the medium and type of delivery mechanism to be used to deliver the contents of the content segment 1810 to the recipient. The message envelope is repeated at 1820 for the particular segment size IDs, times sent, etc., while the magic number and segment count are kept constant at 1830.

Referring to FIG. 13B, the particular available message segment attributes 1790 are illustrated in this exemplary embodiment. A "notify on delivery" attribute 1840 is set by the RTX 460 to request a subsequent session of the RTX to notify the sender on delivery of the message. A "message purge" process provided by this particular message envelope periodically purges messages that exceed a predetermined storage retention time allocated for each type of message. The purge process also informs the broadcast server if it has purged a message that should have been purged based on age, but before it could be picked up by the recipient.

A "reply allowed" attribute 1850 allows the recipient to reply to a message when sent. When the "reply allowed" attribute 1850 is set to zero, it indicates that a mail box owner who sent a general broadcast message does not wish to receive a multitude of replies in response to the general broadcast with a large distribution list. The sender of the general broadcast in preferred embodiments thus has the option of setting the reply allowed attribute to zero. A "private message attribute" 1860 disables the forwarding of a message past the recipient, and an "urgent" attribute 1870 marks the message as urgent and ensures that messages so marked are presented to the recipient before any other messages are presented.

It will be recognized that several types of vectors may be used to manipulate the envelopes and messages as described above. These include, but are not limited to, vectors to create, append, forward, save, purge, pick up, reply, broadcast, and vectors which provide other actions which are particularly used to manipulate mailboxes, envelopes, and other messages. Since the arrangement of vectors to manipulate the envelopes and messages are dynamically programmable by APE 480 and APEX 490 into AST 470, the entire personality of an integrated voice processing system provided in accordance with the present inventions is fully configurable and changeable. Thus, a voice processing system so described may be designed to emulate any existing menu structure, or a completely new menu structure for a system particularly defined.

In addition to the vector manipulation of envelopes and messages, RTX 460 preferably reads information into an out-of-message envelope using voice windows as described herein. In order for a caller to receive a message, a first vector preferably opens the envelope, and a second vector reads information from a message envelope into a voice window. The information in the voice window is then available for processing by all other vectors that are used to develop voice processing applications. Envelopes and messages may also be created using windows to collect information from the caller, or the data communication server 440 and SQL server 510. Voice processing systems provided in accordance with the present invention simply convert voice messaging into an interactive voice response application with the use of a message envelope as an object that is manipulated by vectors wherein the information contained in the envelope and the message are available to all other vectors as the information is obtained from, or bussed to, the data communications server 440 or SQL server 510. In accordance with the present invention, the vectors themselves need not be concerned with the details of the particular VM, facsimile, E-mail, or other voice processing function, but provide a structure for accomplishment of these functions and the relationship between these functions which are stored on a relational database or other system.

Integration of Voice Transaction Vectors

In the case of a collection of vectors which implements voice mail, this set or vectors may be integrated in the same application logic state table with a set of vectors that implements other database or external computer based transaction processes using the data communications server 440 or the SQL server 510. In this fashion, several sets of vectors may be preferably combined to form a tightly integrated, multiple application voice processing system providing efficient interfaces. Examples of these applications include a sophisticated call director, a VM system, an IVR system, a facsimile store forward server, and an E-mail server.

. Additionally in preferred embodiments, groups of vectors defining various applications may be bound together in a single application logic state table 470 and a single communication logic state table 430. Alternatively, they may be separated over several communication logic state tables and application logic state tables. The separate logic state tables may then be treated as objects by a vector from within the logic table that is executable, and operates upon a named logic table external to itself. This further permits creation of a library of vectors that is reusable across diverse sets of customers. Since all these applications are under the control of the same RTX 460 and communication logic state tables 430 and application logic state tables 470, information may be freely exchanged between the various applications.

Thus in accordance with voice processing systems described by this invention, a caller may now request information about an account, listen to this information, request a facsimile copy sent to a particular FAX machine, and leave a message for its account representative without ever leaving the system. In addition, the message sent to the account representative takes advantage of the fact that the caller's account record located on a mainframe using the data communications server 440 contains the identification of the account representative and thus, the caller is never asked to identify the representative nor is the message ever deposited in a general mailbox for manual resorting and redirection. The message also now contains a customer identification, a summary of the caller's account status at the time the message was created, and identifies the fact that the caller requested a facsimile copy of the account information.

Should the representative wish to leave a message for the caller, the customer account information is retrieved into a window, and the "create a mailbox" vector is called to create an on-demand mailbox for the customer if one does not exist so that the message is then deposited into an envelope in a mailbox. The on-demand mailbox is then treated as a regular mailbox until it is aged and purged from the system. On the next call, the caller will then be informed of the waiting message and given an option to be forced to pick up the message before being given any other information.

Thus, it can be seen that a multitude of options and possibilities are opened up by the integration of voice mail, facsimile, and E-mail, with interactive voice response functions using vectors, voice windows, and SQL processing provided in accordance with the present invention and described by this exemplary embodiment. The personality of the voice processing system so described is entirely dependent upon an arrangement which forces the callers to clear their mailboxes before being given further information about their accounts. Other arrangements could, for example, simply inform the callers of the waiting messages, while other examples would require the callers to make a specific selection to check if they have any messages. An endless combination of vectors is therefore possible, and an endless number of applications of seamless integration is provided in accordance with the present invention. Such results have not heretofore been achieved in the art and satisfy a long-felt need for integration of voice processing functions.

There have thus been described certain preferred embodiments of voice processing telecommunications systems provided in accordance with the present invention. While preferred embodiments have been described and disclosed, it will be recognized by those with skill in the art that modifications are within the true spirit and scope of the invention. The appended claims are intended

to cover all such modifications.

CLAIMS: What is claimed is:

[*1] 1. A voice processing system for providing voice transactions through a telecommunications line comprising:

a general-purpose, digital computer adapted to communicate through the telecommunications line with an outside environment;

storage means interfaced to the general-purpose digital computer for storing at least one object having a state associated therewith that is modifiable and that will produce a voice transaction event upon modification;

at least one vector protocol that can be activated by a user in the outside environment through the telecommunications line for operating on the at least one object to produce the change in the object's state; and

an interface adapted to convert user commands input to the voice processing system through the telecommunications line to activate the vector protocol and alter the object's state.

[*2] 2. The voice processing system recited in claim 1 wherein the storage means is a relational database.

[*3] 3. The voice processing system recited in claim 2 wherein the system further comprises run-time executive means on the general-purpose, digital computer for controlling the voice processing system when the voice processing system is in communication with the user through the telecommunications line.

[*4] 4. The voice processing system recited in claim 3 further comprising logic state table means for storing application logic state tables which are used by the run-time executive means and which provide instructions to the run-time executive means to operate the voice processing system.

[*5] 5. The voice processing system recited in claim 4 further comprising editor means for generating the application logic state tables that provide instructions to the run-time executive means to operate the voice processing system.

[*6] 6. The voice processing system recited in claim 5 further comprising server means in communication with the run-time executive means for communicating with external systems in the outside environment and enabling the run-time executive means to interact with the relational database.

[*7] 7. The voice processing system recited in claim 6 wherein the server means further comprises:

a data communication server adapted to interface the voice processing system to external systems in the outside environment; and

an interface server for providing communications protocols to the relational database to enable the run-time executive means to interact with the relational database.

[*8] 8. The voice processing system recited in claim 7 further comprising:

alarm means in communication with the interface server for triggering a system alarm and managing system activity provided by the run-time executive means; and

alarm storage means connected to the alarm means for logging messages generated by the alarm means.

[*9] 9. The voice processing system recited in claim 8 further comprising operational server means in communication with the relational database and the run-time executive means for generating voice processing transactions on command from the run-time executive means as instructed by the application logic state tables.

[*10] 10. The voice processing system recited in claim 9 wherein the operational server means comprises:

an outcall server which is driven by interface statements from the interface server for generating outcall lists to select telecommunications paths for outcalling by the run-time executive means; and

a broadcast server for sorting and distributing information retrieved from the relational database by the voice processing system.

[*11] 11. The voice processing system recited in claim 10 wherein the interface server, data communications server, outcall server, and broadcast server exchange information with the run-time executive means according to at least one voice window.

[*12] 12. The voice processing system recited in claim 11 wherein the voice window comprises a protocol which organizes and manages information exchanges between the servers and system user to provide vector state transitions which produce voice transactions events in response to user input.

[*13] 13. A voice processing system of the type utilizing vectored-state machine architecture comprising:

digital input means for inputting user data over a telephone line;

general-purpose computer means interfaced with the digital input means for processing the user data that is input through the digital input means and for controlling the voice processing system in response to the user data to obtain a voice transaction event;

interface means operatively coupled to the general-purpose computer means for converting the user input data to an interface query;

storage means operatively coupled to the general-purpose computer means through the interface means for storing information that is used by the general-purpose computer means to process the user data to obtain the voice transaction event in response to the interface query;

at least one voice processing object having a state stored on the storage means for producing the voice transaction event when the state is changed; and

at least one vector protocol stored on the storage means for responding to the interface query to change the object's state and produce the voice transaction event.

[*14] 14. The voice processing system recited in claim 13 wherein the digital input means is a TOUCHTONE telephone.

[*15] 15. The voice processing system recited in claim 14 wherein the storage means is a mainframe computer in an outside environment to the voice processing system.

[*16] 16. The voice processing system recited in claim 14 wherein the storage means is a relational database operatively coupled to the general-purpose computer means.

[*17] 17. The voice processing system recited in claim 16 further comprising:

run-time executive means on the general-purpose computer means for controlling the voice transaction event;

logic state table means for storing application logic state tables used by the run-time executive means to provide instructions to the run-time executive means to operate the voice processing system;

editor means for generating the application logic state tables that provide instructions to the run-time executive means to operate the voice processing system; and

server means in communication with the run-time executive means for communicating with external systems in an outside environment and enabling the run-time executive means to interact with the relational database.

[*18] 18. The voice processing system recited in claim 17 wherein the server means comprises:

a data communications server adapted to interface the voice processing system to external systems in the outside environment; and

an interface server for providing communications protocols to the relational database to enable the run-time executive means to interact with the relational database.

[*19] 19. The voice processing system recited in claim 18 further comprising:

alarm means in communication with the interface server for triggering a system alarm and managing system activity in response to the run-time executive means; and

alarm storage means connected to the alarm means for logging messages generated by the alarm means.

[*20] 20. The voice processing system recited in claim 16 further comprising:

operational server means in communication with the relational database for generating the voice transaction event;

run-time executive means for controlling the voice transaction event on command from the operational server means; and

editor means for generating application logic state tables which provide the commands to the run-time executive means.

[*21] 21. The voice processing system recited in claim 20 wherein the operational server means comprises:

an interface server for generating interface statements in response to user input;

an outcall server which is driven by the interface statements for generating outcall lists to select telecommunications paths for outcalling by the run-time executive means; and

a broadcast server for sorting and distributing information retrieved from the relational database by the voice processing system.

[*22] 22. A method of performing voice processing functions in a voice processing system of the type having a general-purpose computer-based control system, utilizing a digital input device and a telecommunications line comprising the steps of:

receiving digital data from a user over the telecommunications line and storing the digital data in the general-purpose computer-based control system;

initializing a state vector function stored in a memory in the general-purpose computer-based control system with the digital input received from the user;

operating the state vector function on a systemdefined object having an initial state associated therewith which is stored on a database associated with the general-purpose computer-based control system, thereby modifying the object's state;

generating a user-recognizable event which is realized by the state vector function's modification of the object's state; and

outputting the user-recognizable event generated by the state vector function's operation on the object.

[*23] 23. The method of voice processing recited in claim 22 wherein the receiving step further comprises the steps of:

providing a run-time executive on the general-purpose computer-based control system; and

controlling the voice processing system with the run-time executive when the voice processing system is in communication with a user through the telecommunications line.

[*24] 24. The method of voice processing recited in claim 23 further comprising the step of generating application logic state tables to be used by the run-time executive and which provide instructions to the run-time executive to operate the voice processing system.

[*25] 25. The method of voice processing system recited in claim 24 further comprising the step of interfacing communications protocols to the database to enable the run-time executive to interact with the database in response to a user's command to modify the object's state.

[*26] 26. The method of voice processing recited in claim 25 further comprising the step of generating a voice window to provide a protocol which organizes and manages information exchanges between the communications protocols and the user to provide vector state transitions which produce voice transactions events in response to the user's input.

[*27] 27. The method of voice processing recited in claim 26 wherein the object, the state associated with the object, and the state vector function are stored on the database in the general-purpose computer-based control system.

[*28] 28. The method of voice processing recited in claim 27 further comprising the step of converting user input to the general-purpose computer based control system into standard queries corresponding to commands input from the user.

[*29] 29. The method of voice processing recited in claim 28 wherein the database is a relational database adapted to respond to the standard queries converted from the user input.

[*30] 30. A method of integrated on-line voice processing with a general-purpose computer which is interfaced to a telecommunications line comprising the steps of:

identifying at least one object having a state in response to a user command wherein the object is adapted to provide a voice processing transaction to a user that accesses the general-purpose computer through the telecommunications line;

acting on the object with a vector protocol adapted to change the state of the object in response to the user's command;

outputting at least one event in response to the changed state of the object after the object has been operated on by the vector protocol;

initializing the voice processing transaction in response to the event; and

communicating the voice processing transaction to the user through the telecommunications line.

[*31] 31. The method of integrated on-line voice processing recited in claim 30 further comprising the steps of:

controlling the voice processing transaction in response to a user input to the general-purpose computer with a run-time executive;

· generating application logic state tables that provide instructions to the run-time executive to control the voice processing transaction;

generating the voice processing transaction to the user on command from the run-time executive as instructed by the application logic state tables which are generated; and

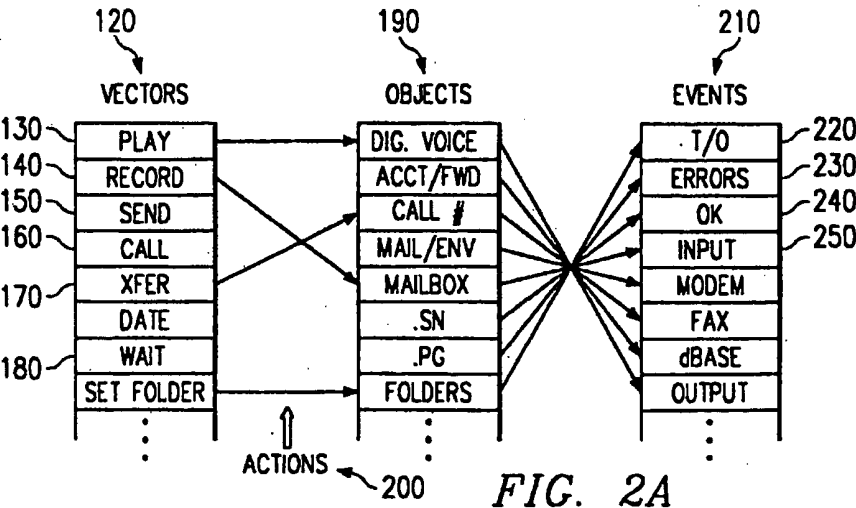
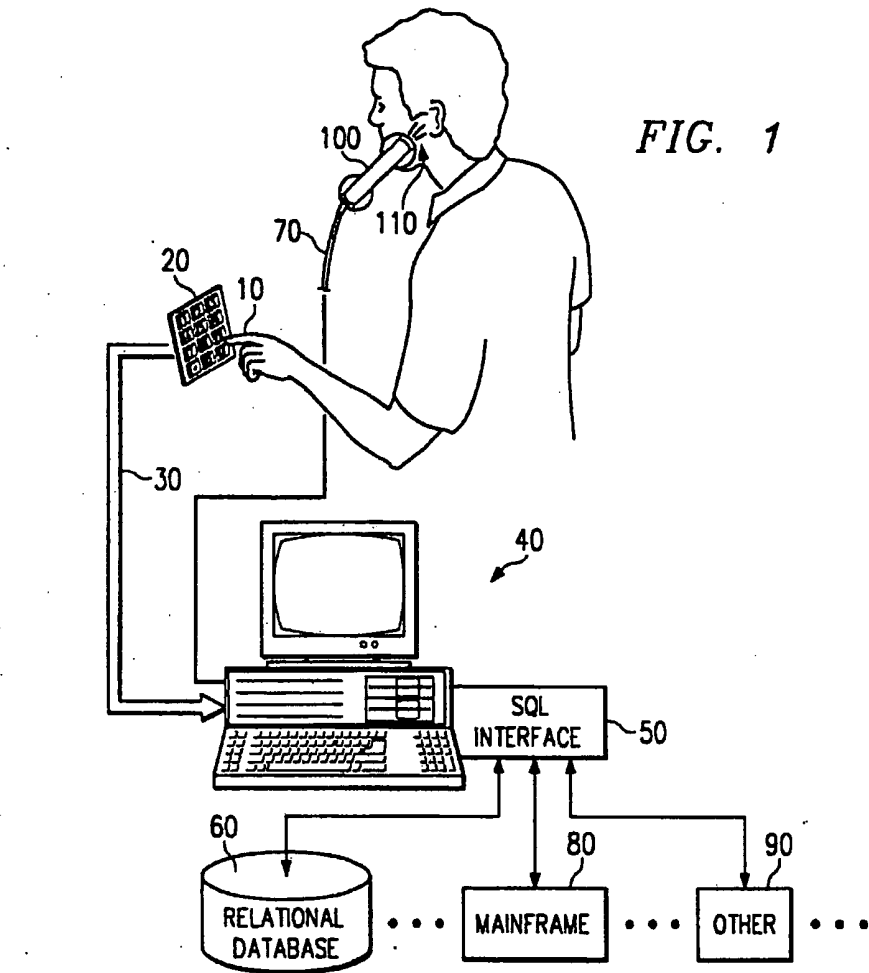
interfacing the voice processing transaction with standard queries after the voice processing transaction has been initialized.

[*32] 32. The method of integrated on-line voice processing recited in claim 31 wherein the object, vector protocol, and event are stored on a storage device.

[*33] 33. The method of integrated on-line voice processing recited in claim 32 wherein the storage device is a relational database.

[*34] 34. The method of integrated on-line voice processing recited in claim 33 wherein the relational database is adapted to respond to standard queries input to the relational database through the general-purpose computer.

[*35] 35. The method of integrated on-line voice processing recited in claim 34 further comprising the step of converting user input to the general-purpose computer to standard queries which are recognizable by the relational database and which enable the vector protocol to operate on the object to produce the state change achieved by the object.



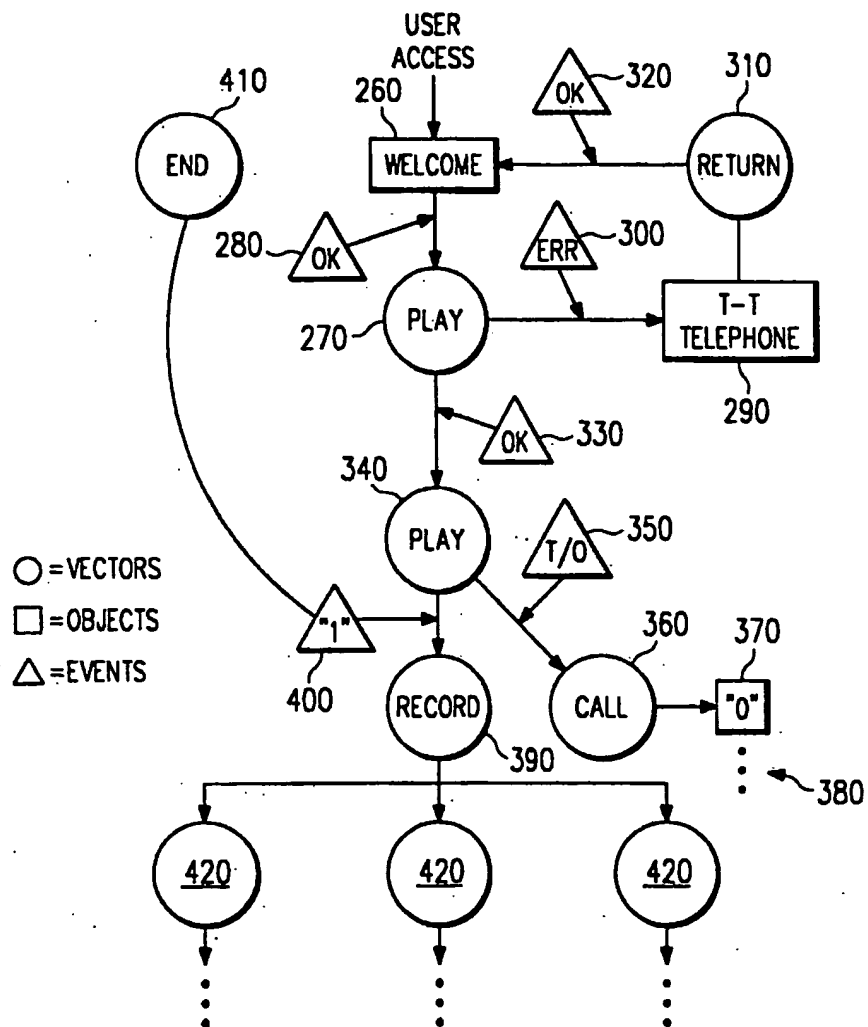


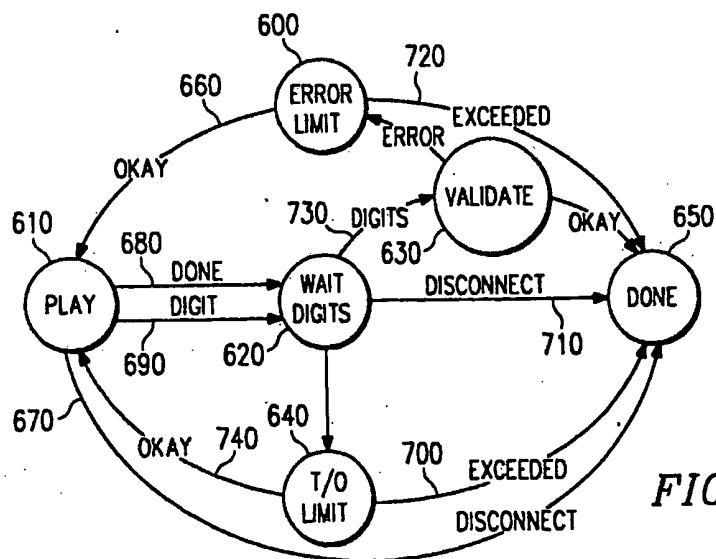
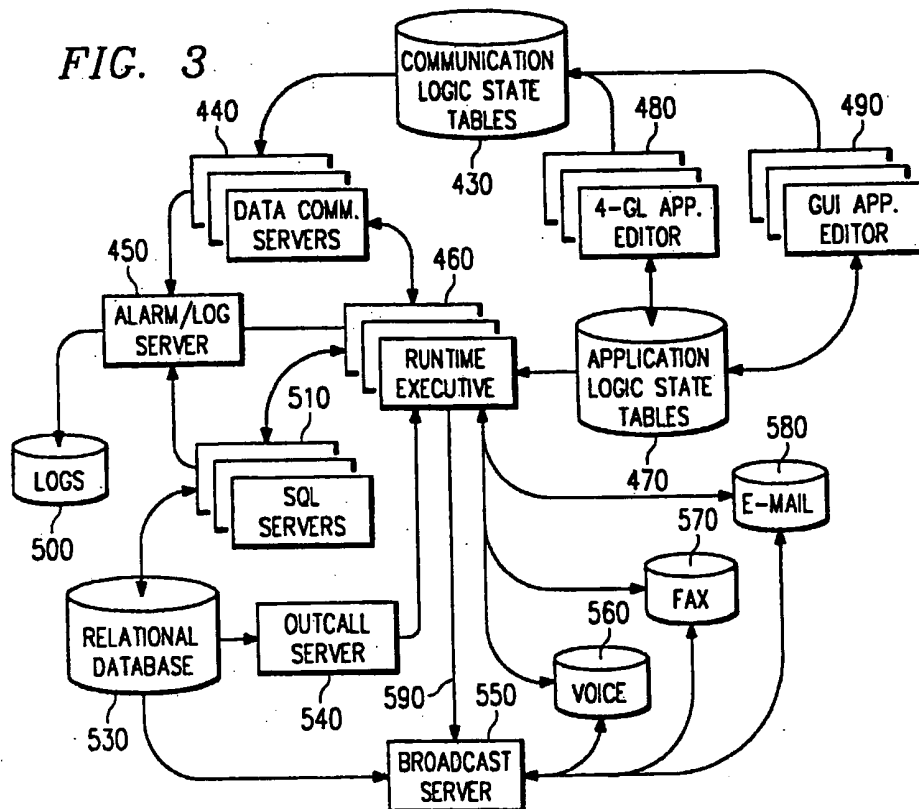
FIG. 2B

U.S. Patent

Oct. 19, 1993

Sheet 3 of 6

5,255,305



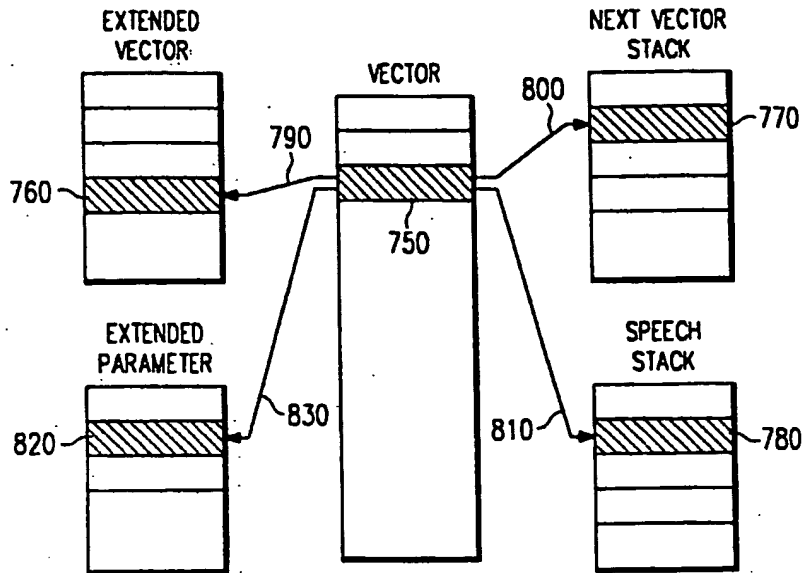


FIG. 5A

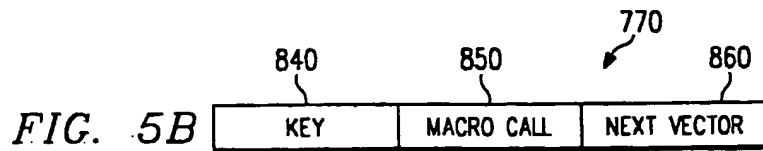


FIG. 5B

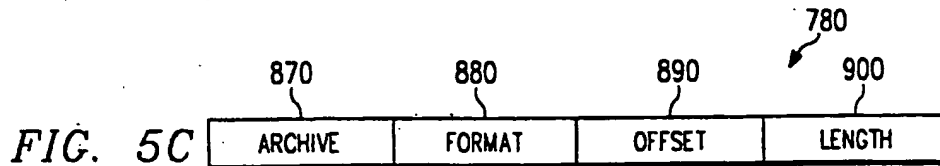


FIG. 5C

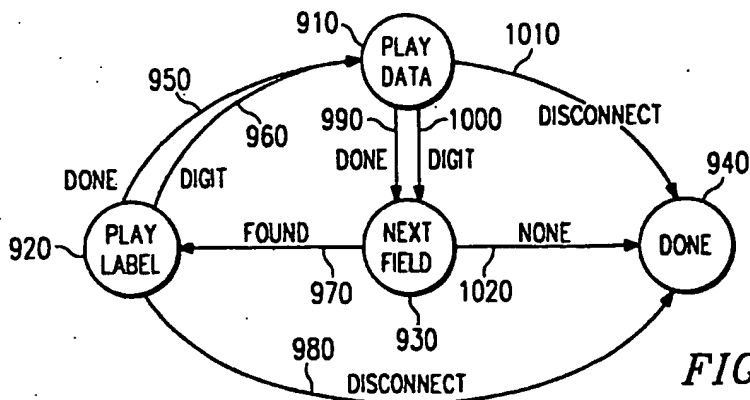


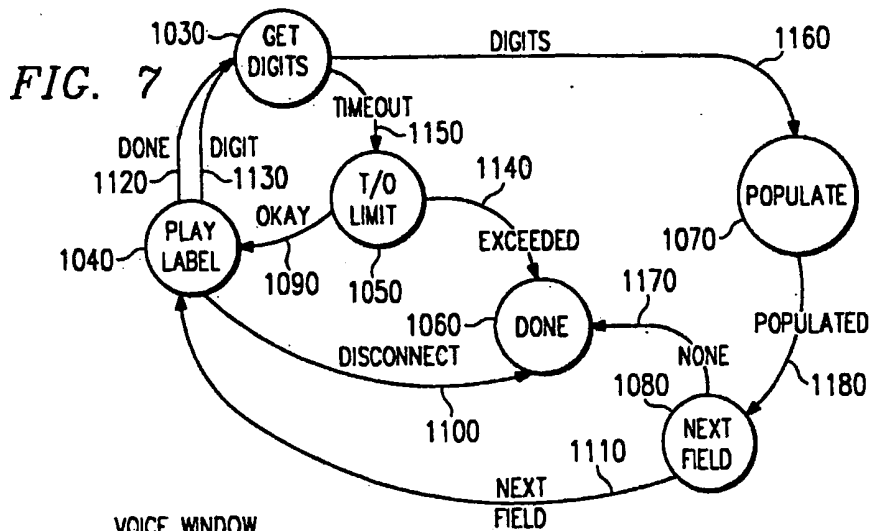
FIG. 6

U.S. Patent

Oct. 19, 1993

Sheet 5 of 6

5,255,305



VOICE WINDOW
FIELD TYPES

1300	CHARACTER
1310	FILLER
1320	TIME STAMP
1330	UNIQUE FILE NAME
1340	FORMATTED DATE/TIME
1350	CONSTANT
1360	REFERENCE
1370	COPY
1380	SEQUENCE

FIG. 9A

VOICE WINDOW
ENUNCIATION TYPES

1390	DATE
1400	MONEY
1410	NUMERIC
1420	PAIRED NUMERIC
1430	PHRASE
1440	TIME
1450	PERCENT
1460	NONE

FIG. 9B

VOICE WINDOW
FIELD ATTRIBUTES

1480	VERIFY
1490	PARTIAL INPUT
1500	CONTINUE ON ERROR
1510	NON-VOLATILE

FIG. 9C

VOICE WINDOW SCHEMA

1190	WINDOW NAME
1200	FIELD NAME
1205	FIELD TYPE
1210	INPUT SIZE
1220	VOICE LABEL
1230	CROSS WINDOW NAME
1240	CROSS FIELD NAME
1250	VALIDATION FUNCTION NAME
1260	CONVERSION FORMAT
1270	ENUNCIATION TYPE
1280	INPUT DELIMITERS
1290	FIELD ATTRIBUTES

FIG. 8

COMPILED VECTOR
ELEMENT

1520	VECTOR NAME
1530	TYPE
1540	SUB-TYPE
1550	PARAMETER
1560	EXTENDED PARAMETER
1570	EXTENDED VECTOR
1580	NEXT VECTOR STACK
1590	SPEECH STACK

FIG. 10

U.S. Patent

Oct. 19, 1993

Sheet 6 of 6

5,255,305

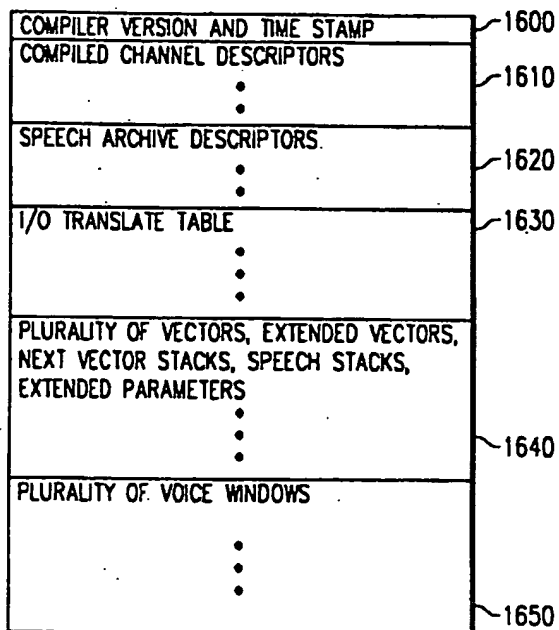


FIG. 11

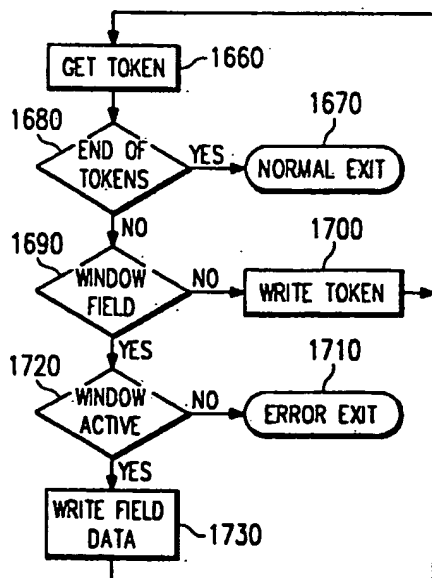


FIG. 12

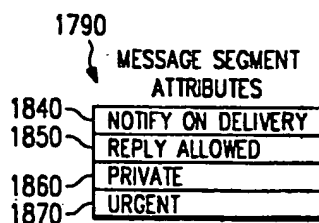


FIG. 13B

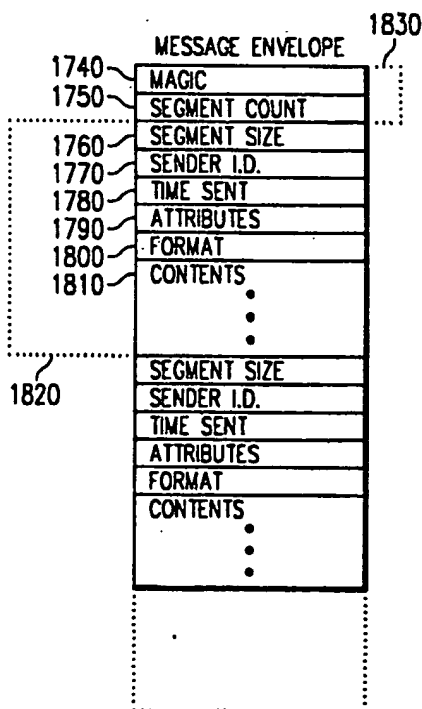


FIG. 13A

THIS PAGE BLANK (USPTO)

Pat. No. 5870464 printed in FULL format.

5,870,464

<=2> GET 1st DRAWING SHEET OF 11

Feb. 9, 1999

Intelligent information routing system and method

ENTOR: Brewster, James A., Plano, Texas
 ramanian, Srikanth, Plano, Texas
 e, Thomas D., Allen, Texas
 , Gene W., Plano, Texas
 nnick, Gary L., Plano, Texas

SIGNEE-AT-ISSUE: AnswerSoft, Inc., Plano, Texas (02)

SIGNEE-AFTER-ISSUE: Date Transaction Recorded: Feb. 22, 1999
 SIGNMENT OF ASSIGNOR'S INTEREST (SEE DOCUMENT FOR DETAILS).

VOX CORPORATION 6 TECHNOLOGY PARK DRIVE WESTFORD, MASSACHUSETTS 01886
 el & Frame Number: 009773/0847

PL-NO: 747,831

LED: Nov. 13, 1996

NT-CL: [6] H04M 3#00

S-CL: 379#219; 379#201; 379#220; 379#229; 395#685

L: 379;395

SEARCH-FLD: 379#201, 219, 220, 221, 93.29, 211, 230, 229, 265, 309; 395#500,
 585, 680

REF-CITED:

	U.S. PATENT DOCUMENTS	
4,747,127	5/1988 * Hansen et al.	379#94
4,878,240	10/1989 * Lin et al.	379#67
5,168,446	12/1992 * Wiseman	705#37
5,202,963	4/1993 * Zellely	395#325
5,206,903	4/1993 * Kohler et al.	379#309
5,289,368	2/1994 * Jordan et al.	364#401
5,404,528	4/1995 * Mahajan	395#685
5,546,452	8/1996 * Andrews et al.	379#219
5,590,188	12/1996 * Crockett	379#225
5,604,896	2/1997 * Duxbury et al.	395#500
5,627,888	5/1997 * Croughan-Peeren	379#201
5,655,015	8/1997 * Walsh et al.	379#201
5,689,799	11/1997 * Dougherty et al.	455#2
5,727,156	3/1998 * Herr-Hoyman et al.	395#200.49

PRIM-EXMR: Zele, Krista

Pat. No. 5870464, *

EXMR: Wolinsky, Scott

REP: Baker & Botts, L.L.P.

TERMS: handle, telephony, route, variable, string, script, htimehandle, ger, simulator, window, hdbhandle, parameter, destvariable, database, f, println, engine, lookahead, invalid, interflow, message, digit, le, declaration, syntax, switch, external, iir, manager, server

:
An intelligent information router system comprising a telephony controller
pled to a private branch exchange through a link interface. The telephony
troller may communicate with a handle manager and a script interpreter
ine. The telephony controller may receive information from the link interface
arding telephone calls being placed to the private branch exchange. The
ephony controller may initiate actions with the script interpreter engine
t access information stored in a database through a database controller. In
sponse to action of the script interpreter engine, the telephony controller
y instruct the private branch exchange to route the call to an appropriate
cation within a company depending on the information received by the private
anch exchange through automatic transmission of data or interaction with the
lling party.

OF-CLAIMS: 42

MP-CLAIM: <=17> 1

OF-FIGURES: 26

DRWG-PP: 11

SUM:

RELATED APPLICATIONS

This application claims the benefit of U.S. Provisional application Ser. No.
60/006663, filed Nov. 13, 1995.

TECHNICAL FIELD OF THE INVENTION

This invention relates in general to the fields of telecommunications and
data processing and more particularly to an improved intelligent information
routing system and method.

BACKGROUND OF THE INVENTION

Advances in interactive voice response systems and private branch exchanges
have allowed for the development of systems which can interact with calling
parties to solicit information in an automated fashion. In addition, modern
database technology can allow for the characterization of a population in
extremely fine detail. Finally, with the development of computer telephony
interfaces, private branch exchanges and other private switching systems can be
accessed and, in part, controlled by efficient and inexpensive personal

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

puters and minicomputers. While all of these building blocks are in place, the work has been done to integrate these facilities to allow the use of the amounts of information stored in a company's database to be used to control access to the company's staff through the telecommunications interfaces.

Accordingly, need has arisen for a system which responds in an intelligent fashion to information provided by a telecommunications system or other data provided by other systems to perform tasks in an automated fashion responsive to the information received.

SUMMARY OF THE INVENTION

In accordance with the teachings of the present invention, an intelligent information router system and method are provided that substantially eliminate or reduce problems associated with prior systems and methods. According to one embodiment of the present invention, an intelligent information router system is provided that comprises a telephony controller which is coupled to a private branch exchange through a link interface. The telephony controller communicates with a handle manager and a script interpreter engine. Telephony controller receives information from the link interface regarding telephone calls being placed to the private branch exchange. Telephony controller initiates actions within the script interpreter engine that, in turn, access information stored in the database through a database controller. Responsive to the action of the script interpreter engine, the telephony controller can then instruct the private branch exchange to route the call to an appropriate location within a company depending upon the information received by the PBX through automatic transmission of data or interaction with the calling party.

DESCRIPTION OF THE DRAWINGS

A more complete understanding of the teachings of the present invention may be acquired by referring to the accompanying FIGURES in which like reference numbers indicate like features and wherein:

FIG. 1 is a schematic block diagram of the environment in which an intelligent information router system of the present invention may operate;

FIG. 2 is a schematic block diagram of the internal architecture of the intelligent information router system of the present invention;

FIGS. 3a, 3b and 3c and FIGS. 4a and 4b are flow diagrams illustrating the operation of various components within the intelligent information router of the present invention;

FIG. 5 illustrates a block diagram of a specific embodiment of the intelligent information router system of the present invention;

FIG. 6a-g illustrate the installation screens, options and operations of a user interface to the intelligent information router system of the present invention;

FIG. 7 illustrates the set-up screen, options and operations of the user interface to the intelligent information router system of the present invention.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

ation;
FIG. 8 illustrates agent station screens, options and operations of the user interface to the intelligent information router system of the present invention;
FIGS. 9a-d illustrate simulator screens, options and operations of the user interface to the intelligent information router system of the present invention;
FIGS. 10a-e illustrate testing screens, options and operations of the user interface to the intelligent information router system of the present invention.

DESC:

TAILED DESCRIPTION OF THE INVENTION

FIG. 1 is a schematic block diagram of a telecommunications system 10 which comprises a telephony server 12 which is coupled through a computer telephone interface link to a private branch exchange 14. Private branch exchange 14 is coupled through a plurality of trunk lines to a central office 16. In addition, private branch exchange 14 is connected to telephones 16, 18 and 20 within agent stations 22, 24 and 26, respectively. Agent stations 22, 24 and 26 also comprise agent work stations 28, 30 and 32.

The agent stations 28, 30 and 32 are coupled to a local area network 34. An administration workstation 36 is also connected to local area network 34. In addition, the telephony server 12 is connected to local area network 34 through a network interface 38.

The telephone server 12 is connected to the CTI link through a link interface 40. According to one embodiment of the present invention, the PBX may comprise a G-3 PBX manufactured by AT&T. Under this embodiment, the CTI link would comprise an AT&T ASAI link. The link interface 40 would comprise an ISDN-BRI board within the telephony server 12. Under this embodiment, the telephony server 12 may comprise a Pentium-class personal computer running the Novell network telephony server system.

The telephony server provides a platform for an intelligent information router system 42. The architecture of the intelligent information router system 42 will be described more completely with reference to FIG. 2. However, in general, calls are received from central office 16 to virtual device numbers within PBX 14. The PBX 14 then generates route requests which are transferred via the CTI link to the intelligent information router 42 within telephony server 12. The intelligent information router 42 processes the route request and generates a route select or a request for further information which is transferred back to the PBX 14 through the CTI link. In some circumstances, the PBX 14 can request information from the calling party using an interactive voice response engine 44 resident on PBX 14. The PBX 14 can solicit customer information using the IVR engine 44 on its own or in response to a request for further information from the intelligent information router 42. Eventually, the intelligent information router 42 will finish processing the route request and will instruct the PBX 14 to route the call to one of the particular agent stations 22, 24 or 26. In addition, information is transmitted via the network interface 38 and the LAN 34 to database facilities 46, 48 and 50 within agent workstations 28, 30 and 32, respectively. In this manner, using information

THIS PAGE BLANK (USPTO)

about the calling party, including an automatic number identification information or information which is solicited using the IVR engine 44, the intelligent information router 42 can access a database using predefined scripts to route the call to an appropriate agent and to supply that agent with information about the calling party. In this manner, for example, a sales organization can route a call to the particular agent assigned to a particular client and present the agent with a recent history of sales activity for that client before the agent says the first word to the client.

The administration workstation 36 is used to create new scripts using a simulator system 52. In addition, the database managed by the intelligent information router 42 is also accessed and administered using a database administration system 54 within the administration workstation 36. The intelligent information router 42 is also administered, maintained and modified using a router administration client 56 within the administration workstation 36. As will be discussed herein, great care is taken in the architecture of the intelligent information router 42 to allow for the alteration of scripts used by the router 42 while the system is active. In addition, the database accessed by the router 42 and the agent stations 22, 24 and 26 and the administration workstation 36 is constantly changing. Mechanisms are also in place within the architecture of the intelligent information router 42 to allow for these changes to take place without interfering from the current activity of the system.

FIG. 2 is a schematic block diagram which illustrates the architecture of the intelligent information router 42. As discussed previously, the information router 42 receives route requests and transmits requests for information and route select data to the PBX 14 through a link interface 40. The link interface 40 interfaces with a telephony controller 58 for all telephony related events, data transfers and instructions.

The information router system 42 receives other non-telephony related information as well as database accesses and updates through an external data manager 60 which is coupled to the local area network 34 through the network interface 38. Administration of the telephony controller and the remainder of the information router system 42 occurs through the operation of a router administration server 62 which is also coupled to the network interface 38 and communicates with the router administration client 56 within administration workstation 36 through the communication path formed by network interface 38 and local area network 34.

Events and processes which are managed and performed by the information router 42 are organized and monitored using a handle manager 64. The handle manager 64 communicates with the external data manager 60, the telephony controller 58 and a script interpreter engine 64 and a database controller 66. In general, all telephone calls that are being handled, data which has been accessed and retrieved, scripts which have been initiated, database accesses which have been initiated, files which have been retrieved, strings which have been parsed, and the like, are organized and monitored by creating handles associated with each of these objects. The handle manager 64 maintains a list of all active handles and is accessed by the remaining components of the information router 42 to insure the duplicative effort is not created. In addition, as will be discussed herein, through the careful operation of the handle manager and the remaining components, the system is allowed to operate constantly while scripts are updated, database information is changed, or any other components are altered.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

The script interpreter engine 65 uses a script storage system 68 for the storage of the data files which comprise the script of activities to be performed by the system 42. The script interpreter engine also comprises a time utility 70, a string parse utility 72 and a file I/O utility 74. The time utility 70 is invoked to monitor and commence time-based events. The string parse utility 72 is used to perform string searches and to parse through alphanumeric and character-based data. The file I/O utility 74 is invoked to access data in files within script storage system 68 and other storage systems accessible to the system 42.

The database controller 66 accesses a database engine 76 which in turn accesses physical database storage system 78. The database engine 76 may comprise any suitable database such as a BTree-based system.

In operation, an external event is received by the router system 42 through the telephony controller 58 or the external data manager 60. As described previously, external telephony events are processed by the telephony controller 58 and all other external events are processed by the external data manager 60. In the telephony application described previously, a telephone call received by the PBX 14 creates a route request event which is received by the telephony controller 58. This route request is then processed by first creating a telephony handle using the handle manager 64. The script interpreter engine 65 is then invoked to process the call. The processing of the call may result in a variety of actions by the system 42, including a request for more information from the PBX 14 or an access to the database engine 76 to retrieve information about the calling party or the call. Finally, the telephony event may result in the telephony controller 58 forwarding a route selection command to the PBX instructing the PBX where to route the particular call.

The architecture shown in FIG. 2 is not limited to the telephony application described previously. For example, other events can be input into the system through the external data manager 60 to similarly cause scripts to be invoked through the script interpreter engine 65. For example, other application programs can communicate with the external data manager 60 using direct data exchange mechanisms or network data exchange mechanisms. These data transfers to the external data manager 60 can cause scripts to be invoked. For example, in one application, a separate application program may monitor a particular stock price. The application program can then cause a data transmission to the external data manager 60 when a particular threshold value for the stock is reached. A script could then be invoked using the script interpreter engine 65 to issue a buy or sell command that can be passed out through the external data manager 60 to a separate program that interfaces with a purchasing or selling agent in that stock.

According to another embodiment of the present invention, the same system 42 is also present in each of the agent stations 22, 24 and 26, and particularly the agent work stations 28, 30 and 32, respectively. According to this embodiment, when a particular telephone call is routed to the agent station telephone 16, 18 and 20, a network message informing the agent's workstations 28, 30 and 32 is also routed simultaneously. The network message is received by an external data manager 60 within, for example, the agent station 28. This network message causes a script within the agent workstation 28 to be invoked. This script could cause certain information to be displayed to the agent on the workstation 28. In addition, particular records within the database engine 76 may be retrieved. In this manner, the agent working at agent workstation 28

THIS PAGE BLANK (USPTO)

will simultaneously receive the call on the telephone 16 and will view the selected database information about the client-calling party simultaneously with the call ringing or even before the call rings.

As discussed previously, the administration workstation 36 functions to manage the operation of system 42. The simulator system 52 comprises a complete implementation of the system 42 within administration workstation 36. All of the components shown in FIG. 2 with the exception of the link interface 40 are present within administration workstation 36 and simulator 52. Link interface 40 is replaced with a graphical user interface to the administration displayable on the administration workstation 36. In this way, new scripts can be developed and tested using the simulator 52.

The database administration system 54 is used by the administrator working at administration workstation 36 to access the database engine 76 and the database storage 78 remotely. These accesses are routed through the network interface 38 and the router administration server 62. This same communication path is also used by the administration workstation 36 to download new scripts to the script interpreter engine 65 and the physical script storage facility 68.

The process of the present invention for routing a call using the intelligent information router 42 will now be described. It will be understood by those skilled in the art, however, that other types of information or data can be routed within the scope of the present invention with the intelligent information router 42.

As shown by FIG. 3A, the method of the present invention for routing a call begins at step 100 and proceeds to step 105. At step 105, a route request is received by the telephony controller 58 from the link interface 40. As previously described, the telephony controller 58 continuously monitors the link interface 40 for route requests. In response to the route request, the telephony controller 58 determines at step 110 if a telephony handle exists for the route request.

The preferred method for determining whether a telephony handle exists is shown by FIG. 3B. However, those skilled in the art will understand that other methods of determining whether a telephone handle exists for a route request may be employed in accordance with the teaching of the present invention. As shown by FIG. 3B, the preferred method is a multi-step process that begins at step 250 and proceeds to step 255. At step 255, the telephony controller 58 creates a prototype telephony handle for the route request. Next, at step 260, the telephony controller 58 accesses the list of existing telephony handles stored on the handle manager 64. At step 265, the telephony controller 58 receives the existing telephony handles from the handle manager in binary sequence. The binary number of each telephony handle is determined by the numeric value of the handle's Call ID. Next, at step 270, the telephony controller 58 compares the binary number of the existing telephony handles received from the handle manager 64 with the binary number of the prototype handle created for the route request. If the binary number of an existing handle is the same as the binary of the prototype handle, then a telephony handle already exists for the route request. However, if no existing telephony handle has the same binary value as the prototype handle, then a telephony handle does not exist for the route request and one must be created.

THIS PAGE BLANK (USPTO)

Returning to FIG. 3A, if a telephony handle does not exist for the route request, the NO branch of decisional step 110 leads to step 115. At step 115, the telephony controller 58 creates a telephony handle for the route request. Next, at step 120, the telephony controller 58 assigns the telephony handle a reference count of 1. The telephony controller 58 then stores the telephony handle to the handle manager 64 at step 125. The telephony handle is stored in the handle manager 64 in accordance with its binary value. This is accomplished by calling the existing handles stored in binary sequence and locating the appropriate position for the new telephony handle. Step 125 leads to step 130 wherein the telephony handle is used to determine a script file name.

Returning to decisional step 110, if a telephony handle already exists for the route requests, the YES branch of decisional step 110 also leads to step 130 wherein the telephony handle is used to determine a script file name. At step 130, the telephony controller 58 determines a script file name for the route request. The telephony controller 58 determines the script file name by using the VDN of the route request, which is packaged with the telephony handle. Next, at step 135, the telephony controller 58 uses the script file name to call a script API to the script interpretation engine 65.

Step 135 leads to decisional step 140. At decisional step 140, the script interpretation engine 65 determines if a script handle exists for the script file name. The preferred process for determining whether a script handle exists for the script file name is shown by FIG. 3C. However, those skilled in the art will understand that other methods may be used within the scope of the present invention to determine whether a script handle exists. As shown by FIG. 3C, the preferred method of determination of whether a script handle exists is a multi-step process that starts at step 300 and proceeds to step 305.

At step 305, the script interpretation engine 65 creates a prototype script handle. Next, at step 310, the script interpretation engine 65 accesses the list of existing script handles stored on the handle manager 64. At step 315, the script interpretation engine 65 receives existing script handles from the handle manager 65 in binary sequence. The binary number of a script handle is determined by the value of the script file name. At step 270, the script interpretation engine 65 compares the binary number of each existing script handle with the binary number of the prototype script handle. If the binary number of an existing script handle matches the binary number of the prototype script handle, the script handle already exists. However, if the binary number of the prototype script handle does not match the binary number of an existing script handle, the script handle does not exist and one must be created.

Returning to FIG. 3A, if a script handle does not exist, the NO branch of decisional step 140 leads to step 145. At step 145, the script interpretation engine 65 creates a script handle. Next at step 150, an executable file is created for the script handle. Proceeding to step 155, the script interpretation engine 65 assigns the script handle a reference count of 1. The script interpretation engine 65 then stores the script handle to the handle manager leads to 64 at step 160. Step 160 leads to step 200.

Returning to decisional step 140, if the script handle already exists, the YES branch of decisional step 140 leads to decisional step 165. At decisional step 165, the script interpretation engine determines if the existing script handle is for the most recent version of the script. Whether the script handle is for the most recent version of the script is determined by comparing the

THIS PAGE BLANK (USPTO)

date of the prototype script handle that was created at step 140 with the date of the existing script handle. If the script handle is not for the most recent version of the script, the NO branch of decisional step 165 leads to step 170.

Proceeding to step 185, the script interpretation engine replaces the outdated script handle with the new script handle. However, the old script handle is not deleted. Rather, at step 190, the old script handle is retrieved to ensure that it is not deleted until all current calls using that script handle are complete. Next, at step 195, the reference count of the old script handle is decreased by 1. As a result, when all calls currently using the old script handle are complete, the reference count of the old script handle will become 0. Thereafter, the old script then will be automatically deleted by the handle manager. Accordingly, the present invention allows scripts to be updated and immediately used for calls received thereafter without interruption of current calls using an old script.

After a script handle has been created or an existing script handle is retrieved, the process proceeds to step 200 wherein the script interpretation engine 65 increases the reference count of the script handle by 1. Next, at step 205, the script interpretation engine 65 determines a route select based on the script. At step 210, the script interpretation engine 65 sends the route select to the telephony controller 58, which forwards the route select to the PBX via the link interface 40. The PBX then routes the call in accordance with the route select.

After a call is completed, the telephony controller 58 decreases the reference count of the script handle by 1 at step 215. Accordingly, when a script handle is in use, it has a reference count greater than 1. When a script handle is idle, it has a reference count equal to 1. When a script handle is outdated and thereafter becomes idle, its reference count drops to 0. Script handles having a value of 0 are automatically deleted by the handle manager 64. Script handles having a value of 1 are known by the system to be idle. Script handles having a value of 2 or greater are known to be in use and, even if outdated, will not have the reference count reduced to 0 and thereby be deleted until the current use is completed.

In another aspect, the present invention provides dynamic extensibility in executing scripts. As shown by FIG. 4A, the preferred script execution process of the present invention begins at step 350 and proceeds to step 355. At step 355, an external module is loaded. Next, at step 360, a function of the module is bound in the script. Thereafter, the script will execute the function when called at step 365.

The preferred process carried out by the script when called is shown by FIG. 4B. The process starts at step 400 and proceeds to step 405. At step 405 the script interpretation engine 65 places each individual parameter of the function directly onto the hardware stack. At step 410, the script interpretation engine 65 marks the position of the hardware stack. This will enable the script interpretation engine 65 to later clear the stack.

Next, at step 415, the script calls the external function without parameters, which are already on the hardware stack. At step 420, the bound function is executed on the parameters on the hardware stack.

THIS PAGE BLANK (USPTO)

Next, at decisional step 425, the script interpretation engine 65 determines if the external function cleared the stack. If the function failed to clear the stack, the NO branch of decisional step 425 leads to step 430. At step 430, the script interpretation engine clears the stack. Returning to decisional step 425, if the external function cleared the stack, the YES branch of decisional step leads to step 435 wherein the process is complete.

Example 1 shows script including code that loads, binds and calls an external function in accordance with the dynamic extensibility of the present invention is provided for the benefit of the reader as part of the specification at the end of the description. In the exemplary script, the module "User32" is loaded and the "LineTo" function is bound to the new script word "LineTo". The function takes three parameters that are each integers. Accordingly, the parameters will be loaded directly onto the hardware stack and left for manipulation by the function. Thus, when the function is called, the function will directly manipulate the parameters on the stack to carry out the desired operation.

Example 1

```
global g-nICRLLogLevel as integer
local hUser32Handle, hGDI32Handle
local hDeskTop as integer
local hPen, hPen2, hOldPen as integer
local rad as integer
local cx, cy, icx, icy as integer
local pi, angle, x, y as double
local ntimes as integer
local RAND-MAX as double
local ltime as integer
g-nICRLLogLevel = 0
hUser32Handle = LoadModule("user32", "user32")
hGDI32Handle = LoadModule("gdi32", "gdi32")
if (hUser32Handle < > 0 and hGDI32Handle < > 0) then
if (BindFunction (hUser32Handle, "MessageBoxA",
"MessageBox", "INT", 4)) then
println "Bind on MessageBoxA succeeded !"
else
println "Bind on MessageBoxA failed !"
endif
endif
BindFunction(hUser32Handle, "wprintfA", "sprintf", "INT",
- 1, 0, 1000)
BindFunction(hUser32Handle, "GetDC", "GetDC", "INT", 1)
BindFunction(hUser32Handle, "ReleaseDC", "ReleaseDC",
"INT", 2)
BindFunction(hGDI32Handle, "LineTo", "LineTo", "INT", 3)
BindFunction(hGDI32Handle, "MoveToEx", "MoveToEx", "INT",
4)
BindFunction (hGDI32Handle, "CreatePen", "CreatePen",
"INT", 3)
BindFunction(hGDI32Handle, "SelectObject",
"SelectObject", "INT", 2)
BindFunction(hGDI32Handle, "DeleteObject",
"DeleteObject", "INT", 1)
hdesktop = - 1
hdesktop = GetDC(0)
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
println "hdesktop = " + hdesktop
hPen = CreatePen(0, 3, 0 x 000000ff)
hPen2 = CreatePen(0, 3, 0 x 0000ff00)
hOldPen = SelectObject(hdesktop, hPen)
rad = 150
pi = 3.141592654
RAND-MAX = 0 x 7fff
ltime = time(0)
println "time( ) = " + ltime
srand(ltime)
rand( )
cx = ConvertToDouble(rand( ))
cy = ConvertToDouble(rand( ))
cx = (cx * 1000) / RAND-MAX
cy = (cy * 700) / RAND-MAX
# icx = ConvertToInt(cx)
# icy = ConvertToInt(cy)
for ntimes = 0 to 10 step 1
println "ring no. " + ntimes
if ((ntimes / 2) = ((ntimes + 1) / 2)) then
SelectObject(hdesktop, hPen)
else
SelectObject(hdesktop, hPen2)
end if
for angle = 0.0 to 2*pi step pi/60.0
x = rad * sin(angle) + cx
y = rad * cos(angle) + cy
if (angle = 0.0) then
MoveToEx(hdesktop, ConvertToInt(x),
ConvertToInt(y), 0)
end if
LineTo(hdesktop, ConvertToInt(x), ConvertToInt(y))
next
next
SelectObject(hdesktop, hOldPen)
ReleaseDC(0, hdesktop)
DeleteObject(hPen)
DeleteObject(hPen2)
# MessageBox(0, "Cool !!!", "Cool !!", 0)
end if
```

In addition to the detailed description of set forth above, Design and User's Guide documentation directed toward a specific embodiment of the present invention (Intelligent Information Router) is provided below. This document is provided for the convenience of the reader and does not limit the scope of the invention to that embodiment.

Design Document for Intelligence Information Router Intelligent Information Router (IIR)

Introduction

THIS PAGE BLANK (USPTO)

This document presents the design of AnswerSoft, Inc.'s Intelligent Information Router. The design description begins with an overview of the IIR design and continues with detailed descriptions of the subcomponent units which implement the design. This document ends with a closure section discussing extensibility issues of IIR for the future.

The IIR design is based on two overriding principles. First, extensibility is of utmost importance. Second, field maintenance requires that field upgrades be easy and cost effective to implement. This can mean, among other things that field upgrades should be possible without stopping the system. To support these requirements, the design uses a multiple module single level communication bus.

Four main components, shown in FIG. 5 and explained below, make up the IIR: the Intelligent Call Routing Language scripting engine (ICRL), the Intelligent Call Routing Telephony module (ICRTEL), the Intelligent Call Routing Customer Database module (ICRDB) and the Intelligent Call Routing General Utilities module (ICRMSC). Each module is responsible for the creation of information sets (data objects) which pertain to the module's subject. The information sets are created on behalf of ICRL. Once created and initialized handles to the information packets are given to ICRL which then manages access and destruction of the information sets. This is discussed in detail in the ICRHANDLE API Specifications. The information sets can be accessed and manipulated through the dynamically bound API's exposed to ICRL (and therefore available to scripts) by any of the functional modules.

In order to support internal Q/A, field technical support and user level debugging, IIR implements a multiple level Audit Trail execution dump facility. Audit Trail files provide a complete log of the execution path of a script and the code executed by the script. It is instructive, at this point, to present a short example of a call route prior to detailed descriptions of the modules which perform the work.

1. An incoming call arrives at the PBX
2. ICRTEL receives a corresponding event via the telephony system communicating with the CTI link.
3. ICRTEL creates (allocates) a data information packet which holds all known information about the call, including ANI, DNIS, Prompted Digits, etc.
4. ICRTEL calls a fixed (known) entry point inside the ICRL module which begins the call routing process.
5. The ICRL interpreter runs the relevant scripts to process the call. In doing so, the scripts can and will use the functions exported by the various dynamically bound components.
6. Using the function calls exposed to the ICRL interpreter, the scripts call ICRDB to access a customer database to either find or create a new customer record.
7. If a customer record is found, the script receives a handle to the record in which that customer's data is cached (in memory).

THIS PAGE BLANK (USPTO)

8. The script then calls another ICRDB function to look up the most recent agent to service the customer.

9. Finally the script calls an ICRTTEL function to route the call to the agent's phone.

It is important to understand that the functionality used by the script, i.e. function calls, are not defined in the language of ICRL. They are defined by the functional modules, such as ICRTTEL and ICRDB, and provided to the ICRL engine at system start time, or any time thereafter in the form of other (more basic) script language function calls. This is known as dynamic binding and language extension. ICRL depends heavily on dynamic binding to extend the functionality of its scripting language, without itself requiring changes. This dynamic extension of the ICRL interpretive language allows extreme flexibility to functionality upgrades as well as field repairs to bugs.

The Intelligent Call Router Language (ICRL)

Overview

ICRL is a free-form, interpreted language which has structured programming features. All functionality beyond language definition will be handled externally through script function calls, provided to ICRL, by other modules within the system. ICRL defines nothing about the services it gains from other modules of IIR, but requires them in order to carry out any interesting actions such as a call route. ICRL is built as a stand alone Windows DLL or a Novell NLM. The ICR prefix of the name is a misnomer of the project which bore out this design. ICRL is useful to any task automation process where information feeds can be attached and script functionality defined to access and manipulate the information.

Running Environment

All external modules of IIR will communicate only indirectly with the ICRL engine. ICRL provides a couple of intrinsic functions through which scripts can load an external module and then bind script function names to functions within the external module. ICRL adds these functions and the syntactical names to its table of known tokens in the interpreter.

This dynamic function extension mechanism alone provides virtually infinite extensibility of ICRL with no coding changes to ICRL itself. It is usually a good practice to run a set of scripts at system startup time which perform the operations of loading the various external IIR modules and binding all the functions each of these modules provide. This not only speeds up the process of a script later accessing an external function, but also simplifies it because the script writer need not bother loading the external module or binding the function that is needed. Once the function tables are initialized (at startup), the ICRL engine will go into a listening mode waiting for incoming script run requests.

ICRL Protocol

The run-time behavior of the ICR system with the ICRL engine at the core will be controlled through the following protocol.

THIS PAGE BLANK (USPTO)

ICRL Startup Phase: At system startup time, the ICRL engine will initialize itself which might include location and loading of global scripts.

ICRL Accept Request Phase: After step 1, the ICRL engine will go into a listening mode where it will wait for command process requests from any of the modules. The individual modules then may call ICRL API any time in this mode.

The ICRL engine provides two simple calls in its API either one of which external IIR modules can use to run scripts directly. They are ICRLRunScript() and ICRLRunScriptEx(). There are two other functions provided by ICRL that can be used together run scripts in a two step process: ICRCreateScriptThread() and ICReExecuteScriptThread(). ICReExecuteScriptThread() and ICRLRunScriptEx() accept variable number of arguments which are handed to the script as command line-like arguments.

For example, when an incoming call reaches ICRTTEL, ICRTTEL creates or updates the related information packet and calls ICRCreateScriptThread() and ICReExecuteScriptThread() with its module ID and the handle which identifies the modified information packet as an argument. The handle contains pertinent information about the call for which the event triggered a script run, such as CallID, ANI, DNIS, etc. The Telephony component creates such a handle for each active call. Functions called from scripts back into the telephony component can use the handle to retrieve information about the call. Handle usage and specifications are discussed in detail in ICRHANDLE API below.

System Startup: At system startup time, the ICRL engine loads a pre-defined script and runs it. This script may, in turn, load and run other scripts using ICRLRunScript() or ICRLRunScriptEx(). Any of the startup scripts can make calls to the two ICRL intrinsic functions: LoadModule() and BindFunction() which allow loading of all external IIR modules and binding of script function names to function pointers within the external module.

The beauty of this design is that the ICRL engine requires no knowledge of the outside world-i.e. which events stimulate script runs and which functions are available during a script execution. The entire system is fully dynamic. This is the key to field upgradability. Since ICRL allows infinite external modules, different versions of the same module can be run at the same time. This allows the IIR to be upgraded while running.

The generation of script execution threads and script caching is handled in a layer built over ICRL and which interfaces to the functional modules of the IIR. This layer also handles issues closely related to the ICRL script engine by providing miscellaneous utilities such as memory management, information packet access referencing and destruction of the information packets when appropriate. One thing to note is that ICRL itself does not (and need not) distinguish between functions provided by this layer and functions provided by other components.

All functions in the function table can assume information packets which they manipulate are handle-based. That is, the allowed parameter types and return types of external functions are pre-defined and cannot be changed or added to. Though this may seem like a severe restriction, it is actually the exact opposite. This implementation frees ICRL from issues with respect to different data types, allowing it to indirectly handle all data types. The implication of is that ICRL does not support data structures or user defined types. If an

THIS PAGE BLANK (USPTO)

ICRL script has to handle fields in a data structure, ICRL has to be given a set of Get/Set functions (in the function table) which work on a handle to that type of structure and return/set the individual fields in that structure. Therefore compatibility issues like Unicode or multi-byte (international character set) compliance do not arise as the script will have external handlers which do the necessary work to provide compliance as necessary.

ICRL Features

The various language constructs which will be supported by ICRL are listed in the IIR user documentation.

Implementation Issues

The ICRL interpreter engine is implemented using the UNIX compiler generation tools Lex and Yacc. Lex and Yacc together form a powerful compiler definition tool set through which syntax for a language definition can be specified in human readable form. The C source code generated by Lex and Yacc is encapsulated in a C/C++ layer that forms the complete parser/interpreter engine.

In addition to the ICRL parser, two other issues are handled by the ICRL module-memory management and string management. Memory management is completely handle based. All functional modules allocate their information packets on behalf of the ICRL engine. These packets are known as handles which relate events to the tied information. Handle aggregation is the technique used to tie the disparate pieces of information with reference counts and the modules that created and can act on the packets.

The ICRL module performs all handle management and owns all handles even though the functional modules create and initialize them. ICRL must own the handles because there may be multiple scripts running and accessing the same information simultaneously. Rather than let each module implement its own referencing strategy, a common strategy is implemented by ICRL. This is explained below in detail in ICRHANDLE API Specifications..

Error Reporting and Audit Trail Logging

ICRL allows two forms of error reporting. First, a syntax pre-parse of scripts will be performed on all scripts. Prior to running a script, whether for simulation purposes or actual implementation, ICRL reads and parses the script in order to build an execution tree. At each point in the script parsing, ICRL knows whether the script is in proper ICRL form and matches syntactical rules. If the rules are broken at any point in a script, ICRL will provide error reports stating the line number and expected syntax at the point the script is in violation and terminate the pre-parse. Termination is not strictly required, as the recursive decent parser of ICRL is capable of continuing, the parse state after a syntax error can't be guaranteed and significant syntax fallout can occur. In such a case, large amounts of errors will be generated which could actually be erroneous errors.

Second, as all modules of IIR require, ICRL will provide audit trail logging of each code entry point entered during the script execution. It is not possible to define, completely, at this point how much audit trail logging will exist in any given entry point. At the least, each entry point will provide two logs. First the entry which logs all parameters passed into the entry point. Second,

THIS PAGE BLANK (USPTO)

the exit point and all parameters and all return values. Any information packet handle will be recursively dumped to the audit trail file.

It is useful to allow levels of audit trailing. The recursive dumps of handles can be expensive in terms of file space, as the information contained in the handle can be large. Therefore, ICRHANDLE objects will be logged based on the bLog member of the handle. This way, a script can turn ICRHANDLE level logging on and off at will to control the size of the audit trail dump.

ICRHANDLE API Specifications

The ICRHANDLE type is the object by which all IIR subsystems transfer their information into and out of the ICRL engine. All subsystems of the IIR must support the use of the ICRHANDLE to hold store and/or frame allocated information packets. The base handle consists of the following members listed below in Table 1:

TABLE 1

struct tagICRHANDLE	
{	
CTINT nModuleID;	This is the unique module identifier
CTINT nReferences;	Counter of entities which are currently referencing this object. If the references ever decrease to zero, the ICRL engine will automatically destroy the handle via the handle's virtual HandleDestroy API.
CTINT bLog;	TRUE if this object should be dumped to the audit trail log. FALSE otherwise. This defaults to FALSE.
int (*pHandleCompareFunc)(ICRHANDLE h1, ICRHANDLE h2);	Pointer to function, provided by the subsystem, which is used by the ICRL engine to compare any two handles belonging to that subsystem.
int (*pHandleDestroyFunc)(ICRHANDLE h);	Pointer to function provided by the subsystem which is used by the ICRL engine to free memory associated with that handle after its lifetime.
};	
typedef struct tagICRHANDLE ICRHANDLEINFOSTRUCT;	
typedef void* ICRHANDLE;	

Making the ICRHANDLE a void* hides the ICRHANDLEINFOSTRUCT data members from modules which do not need to know what an ICRHANDLE is. They treat ICRHANDLES as black boxes. Private modules which do need the ICRHANDLE structure can type cast an ICRHANDLE to the specific structure pointer they want. Any subsystem which registers itself with the ICRL engine must export the following entry points with respect to a handle as listed below in Table 2:

TABLE 2

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
CTINT
HandleCompare(ICRHANDLE
h1, ICRHANDLE h2);

void
HandleDestroy(ICRHANDLE);

void
HandleDump(ICRHANDLE);
```

This entry point compares two handles and returns - 1, 0 or 1 for h1 < h2, h1 == h2 and h1 > h2 respectively.

This entry point cleans up all issues with respect to a handle of information and frees the handle itself.

Dumps the handle to the currently active Audit Trail file.

The ICRHANDLE API will handle all storage and lookup of all handles. The API exposes a high speed lookup mechanism for locating handles based on the contained information and the

HandleCompare entry point. The following functions are exported to other subsystems by the ICRHANDLE API itself and need not be overridden or duplicated as listed below in Table 3:

TABLE 3

```
HandleFind(ICRHANDLE
hLooksLikeThis);

HandleAddToTable(ICRHANDLE
hAdd);

HandleRemoveFromTable
(ICRHANDLE hRemove);
```

Locates a handle in the handle tables via a (possibly temporary) handle filled with enough information to complete calls to the HandleCompare method. At a minimum, the nModuleID must be provided, as well as any information required by the calling module's HandleCompare entry point.

Adds a handle to the table of handles currently known to the system. This method should always be called after handle creation and initialization. Note that the ICRHANDLE API does not call HandleAddRef. This way calls to DecrementReference do not have to distinguish between whether a handle has a reference and if so, is it in the tables. The handle must be fully initialized before calling this entry point.

Removes a handle from the table of handles currently known to the system. This method should be required only rarely handle manager will automatically remove handles from its tables when they are about to be destroyed

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
void HandleDump(ICRHANDLE);           Dumps the base handle portion
                                      of a handle to the current
                                      Audit Trail file.

HandleDestroy(ICRHANDLE               Destroys the handle. This
hDestroy);                           method automatically locates
                                      the correct module and entry
                                      point in that module to call
                                      for proper destruction.

HandleAddRef(ICRHANDLE               This method increases the
hThis);                              reference count on a handle.
                                      Handles will not be destroyed
                                      until the reference count has
                                      dropped to zero or less.

HandleRelease(ICRHANDLE              This entry decrements, by one
hThis);                              the reference count of a
                                      handle. If the reference count
                                      drops to zero, the handle will
                                      be removed from the handle
                                      tables and its destruction
                                      entry point will be called via
                                      DestroyHandle.
```

As a simple example, a telephony subsystem would allocate information blocks which hold interesting information about telephony events similar to the following Example 2:

Example 2

```
struct tagTELINFOSTRUCT
[

    CTINT          nCallID;
    CTINT          nCrossRefID;
    CTCHAR         szDevice[32];
    CTCHAR         szInstrument[32];
    .
    .
    .

];
typedef tagTELINFOSTRUCT TELINFOSTRUCT,
*PTELINFOSTRUCT;
struct tagTELHANDLE
[
    ICRHANDLEINFO BaseHandle;
    TELINFOSTRUCT iTelInfo;
]
typedef tagTELHANDLE TELHANDLE, *PTELHANDLE;
CTINT g-nModuleID;
static FUNCTIONTABLE s-ftTable; /* this gets filled out
somewhere */
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```

CTINT
TelCompareHandle(ICRHANDLE h1, ICRHANDLE h2)
{
    PTELHANDLE p1 = (PTELHANDLE)h1;
    PTELHANDLE p2 = (PTELHANDLE)h2;
    CTINT nCompare = p1 -> nCallID - p2 -> nCallID;
    return nCompare > 0 ? 1 :
    nCompare < 0 ? -1 :
    0;
}

void
TelDestroyHandle(ICRHANDLE hThis)
{
    PTELHANDLE pThis = (PTELHANDLE)hThis;
    /* close files, cleanup net connections, free store */
    UtsMscFree(pThis);
}

ICRHANDLE
RegisterAndMakeOne( )
{
    PTELHANDLE pThis = (PTELHANDLE) UtsMscMalloc(sizeof
    (TELHANDLE));
    g-IRCLModuleID = ICRLRegister(s-ftTable,
    TelDestroyHandle, TelCompareHandle);
    HandleAddToTable((ICRHANDLE)pThis);
    return (ICRHANDLE)pThis;
}

BOOL
TelProcessEventFromPBX(PEVENTTHING pEvent)
{
    TELHANDLE hTest;
    PTELHANDLE pThis;
    memset(hTest, 0, sizeof (hTest));
    hTest.nModuleID = g-TelModuleID;
    hTest -> nCallID = pEvent -> nCallID;
    pThis = HandleFind(&hTest);
    if (!pThis)
    {
        /* make a new handle and fill it up */
        pThis = (PTELHANDLE)UtsMscMalloc(sizeof
        TELHANDLE);
        if (pThis)
        {
            pThis -> nCallID = pEvent -> nCallID
            /* and so on */
            HandleAddToTable((ICRHANDLE)pThis);
        }
    }
    /* do whatever should be done to actually handle the
    event
    * like fill up the structure and whatnot
    */
    return CTIERR-ALL-OK;
}

```

THIS PAGE BLANK (USPTO)

Intelligent Call Routing Telephony Module (ICRTEL)

Overview

The IIR's main purpose at revision level 1.0 will be efficient, intelligent call routing. For this to take place, a complete telephony module must be included which implements the functionality required to perform the route command based on the decision logic in the routing scripts. ICRTEL is this module.

The functionality of this module is rather limited, in that its only job is to provide Get/Set methods on the information packets created to represent active calls, methods to perform route sequences. Routing sequences are defined in the ICRL scripts. The route requests within the scripts are carried out by this module.

The general nature of this module follows:

1. A Call becomes available at the PBX
2. The PBX delivers the call events across the CTI link to the telephony server
3. ICRTEL receives the event message and creates an information packet which holds all pertinent information and history about the call.
4. ICRTEL calls ICRCreateScriptThread() and ICRExecuteScriptThread() to run a script and passes the ICRHANDLE which represents the information packet.
5. The Script makes Database lookups as necessary to determine the correct route
6. The Script calls ICRTEL indirectly through the dynamically bound script functions.
7. ICRTEL performs the desired request, which may be a Get/Set request or a route. A route takes two forms, temporary or terminating. A temporary route is a request to route the call to a VDN in order to collect additional InfoDigits. The script which requests a temporary route does not terminate, rather it is suspended until the VDN collection is complete, after which the script resumes. Prior to resuming the script, ICRTEL fills additional InfoDigit packets in the call information packet corresponding to the call. This additional InfoDigit information is appended to the InfoDigit information already present. Appendage is required in order to allow historical decisions based on the VDN paths during a complete route.

Terminating route requests are routes which are known to have reached the call's final destination. This is representative of a call being sent to a specific agent or agent group. The terminating call route does not, however, terminate the script. It simply terminates the call's ability to be routed to additional sites. The only exception to this rule is if PBX allows a call to be removed from a queue, assuming the call is still in a queue. In such a case, another ICRTEL script API might provide access to that functionality. The

THIS PAGE BLANK (USPTO)

script resumes even after a terminating route request, though, so additional logging might be made or other not-routing commands.

Since the ICRL engine provides for information packet handle management, ICRTel does not attempt to manage the handles it creates. ICRTel inserts, via HandleAddToTables(), handles to the information packets it creates. No other memory and/or handle management is required. Examples of ICRTel API exposed to scripts follows in Example 3.

Example 3

ICRTel API

```
ICRTelGetANI(ICRHANDLE); qwer
```

```
ICRTelSetANI(ICRHANDLE); qwer
```

```
ICRTelGetDNIS(ICRHANDLE); qwer
```

```
ICRTelSetDNIS(ICRHANDLE); qwer
```

```
ICRTelGetCallID(ICRHANDLE); qwer
```

```
ICRTelRoute(ICRHANDLE); qwer
```

```
ICRTelRouteInfoDigits(ICRHANDLE); qwer
```

Intelligent Call Routing Database Module (ICRDB)

Overview

The IIR's main purpose at revision level 1.0 will be efficient, intelligent call routing. For this to take place, a complete database module must be included which implements the functionality required to perform the route command based on customer database information. ICRDB is a generic implementation of customer records.

The first version of ICRDB allowed only simple single field queries. The current version of ICRDB (ICRDBSQL) implements a generic SQL query and update ability. The most important feature of this external module is that it is designed to be independent of the actual database engine used.

ICRDB provides methods to perform queries on the database, find the count of records matching a query and standard forward/reverse traversals (enumeration) of the records. Additionally, within each record, ICRDB provides Get/Set methods per field. This is a difficult issue and may change in design before final shipment of the IIR.

The nature of this module is as follows:

1. A Call becomes available at the PBX
2. The PBX delivers the call events across the CTI link to the telephony server

THIS PAGE BLANK (USPTO)

3. ICRTTEL receives the event message and creates an information packet which holds all pertinent information and history about the call.

4. ICRTTEL calls ICRCreateScriptThread() and ICRExecuteScriptThread() to run a script and passes the ICRHANDLE which represents the information packet.

5. The Script makes a query on the customer database based on ANI.

6. ICRDB runs an SQL statement against the database engine which generates a record set of matches. The records are stored in an ICRHANDLE subclass and the handle is returned to the Script.

7. The Script requests from ICRDB, the number (count) of matching records in the record set.

8. The Script rolls (traverses) through the matching records and calls ICRDB to determine, for instance, the speaking abilities of the customer.

9. The Script decides, after locating the correct record, that a Spanish speaking agent is required.

10. The Scripts requests ICRTTEL to route the call to a Spanish speaking agent.

Since the ICRL engine provides for information packet handle management, ICRDB does not attempt to manage the handles it creates. ICRDB inserts, via HandleAddToTable(), handles to the record sets created during the running of a script. The disadvantage of this is that it is the responsibility of the script programmer to request that ICRL remove the record set handles prior to script termination. No other memory and/or handle management is required. Examples of ICRDB API exposed to scripts follows in Example 4.

Example 4

ICRDB API

ICRDBRunQuery()

ICRDBGetStringFieldValue()

ICRDBSetStringValue()

ICRDBGetNumericFieldValue()

ICRDBSetNumericFieldValue()

ICRDBMoveNextRecord()

ICRDBMovePreviousRecord()

ICRDBMoveUpdateRecord()

Future

THIS PAGE BLANK (USPTO)

The design of ICRL is based on extensibility requirements. The separation of language features from external functions and dynamic binding of functions makes ICRL very flexible. New functions or even whole components and media (information) feeds could be added to the IIR with minimal effort. The usage of Lex and Yacc guarantees not only good maintenance of code but also extensibility of language features for future releases.

User's Guide Documentation for Intelligent Information Router

Contents of the User's Guide is shown by Table 4.

TABLE 4

1-Getting Started

Requirements

Hardware
Software

Getting Help

Understanding Typographical Conventions

Installing the Program

Installing Over Existing Files
Beginning the Installation Routine
Installing the System Modules

Using the Program for the First Time

2-Using the Database Administrator

Setting Up Fields
Defining Primary Key Fields

3-Using the Agent Station

Searching the Database

Using Wild Card Characters

Modifying the Database

THIS PAGE BLANK (USPTO)

- Adding a Record
- Modifying a Record
- Deleting a Record

4-Writing Scripts

- Using the Simulator Window

 - Changing the Appearance of the Simulator

 - Window

- Writing Script

 - Opening a Script
 - Formatting a Script
 - Saving Your Script

- Printing a script

 - Previewing a script
 - Setting up the printer
 - Selecting a printer font

5-Testing and Implementing your Scripts

- Testing Scripts

 - Test Non-Telephony Scripts
 - Testing telephony Scripts
 - Viewing the Results

- Making the Script Available
- Using the VDN Administrator

 - Adding a Script
 - Starting and Stopping a Script
 - Refreshing the VDN Settings
 - Deleting a Script
 - Modifying the VDN Settings

THIS PAGE BLANK (USPTO)

6-Using the Scripting Language

Understanding the IIR Environment

Integrating IIR with the AT&T Switch

Using Operators and Expressions

- Variables (Declaration, Usage)
- Simulating Constants with Variables
- Arithmetic Operators
- Relational and Logical Operators
- Assignment Operators and Expressions
- Script Output and String Operators
- Comments

Control Flow

- Statements and Blocks
- If-Else-Endif
- Select Statement
- Loops: For-Next and Do-Loop Until

Program Structure

- IIR Function Categorization
- Guidelines to Follow

Command Summary

A-Command Reference

- Function ClearRecord
- Function ClearDBHandle
- Function CreateTimeHandle
- Function DestroyDBHandle
- Function DestroyTimeHandle
- Function GetAgentAvailable
- Function GetAgentState
- Function GetAgentTalkState
- Function GetAgentWorkMode
- Function GetAscTime
- Function GetCallingDevice
- Function GetCurrentTime
- Function GetDayOfMonth

THIS PAGE BLANK (USPTO)

Function GetDayOfWeek
Function GetStringFieldValue
Function GetTrunk
Function GetVDN
Function GetYear
Function ICRLAtoi
Function ICRLLeft
Function ICRLMid
Function ICRLRight
Function ICRLStrCopy
Function ICRLStrIndex
Function ICRLStrLen
Function ICRLStrStr
Function InsertRecord
Function MoveNextRecord
Function MovePreviousRecord
Function QueryAgentState
Function RouteFinal
Function RouteMore
Function RouteUnknown
Function RunQuery
Function SetCurrentIVRSets
Function SetDestRoute
Function SetDirectedAgentCallSplit
Function SetNumericFieldValue
Function SetOutgoingUI
Function SetPriorityCall
Function SetRouteSelected
Function GetStringFieldValue
Function SetUserProvidedCode
Function UpdateRecord

B-Sample Scripts
C-Terms and Acronyms
D-Error Codes

IIR Simulator Errors
IIR Database Administration Tool
IIR Agent Tool

Error Codes

Time Command Errors
Database Error Codes
Telephony Error Codes
Miscellaneous Low Level Error Codes

Chapter 1-Getting Started

THIS PAGE BLANK (USPTO)

The Intelligent Information Router (IIR) is a server-based application for routing incoming telephone calls based upon call information and a set of rules.

The IIR application has the following components:

TABLE 5

On the NetWare Server	Route Engine	A collection of network loadable modules:
	*	A script engine determines which scripts to start and subsequent routing/distribution decisions.
	*	A telephony module allows the route engine to talk to the T-server
	*	A database module allows the script engine to communicate with the databaseDatabaseData that you enter into the system.
	*	The database contains 15 text fields and 5 numerical fields.
	*	Module used for writing and testing scripts, viewing output.
	Database Administrator	Module that allows the administrator to label and define the fields in the Agent Station. The administrator also assigns Agent viewing and modifying privileges for all fields.
	*	Module that allows agents to search, view, and (in some cases) modify database information.
	VDN Administrator	Module which matches scripts to Vector Directory Number (VDN) information for the routing engine. It also starts and stops scripts.
	*	
On the Client	Simulator	Module used for writing and testing scripts, viewing output.
	Database Administrator	Module that allows the administrator to label and define the fields in the Agent Station. The administrator also assigns Agent viewing and modifying privileges for all fields.
	Agent Station	Module that allows agents to search, view, and (in some cases) modify database information.
	VDN Administrator	Module which matches scripts to Vector Directory Number (VDN) information for the routing engine. It also starts and stops scripts.

Requirements

Check your system to be sure that you have the hardware and software the Intelligent Information Router needs to operate successfully.

Hardware

The following hardware requirements listed in Table 6 are minimum requirements.

TABLE 6

For your NetWare server	For your client machines
Pentium, 100 MHz	486-33 MHz
5 megabytes free hard	10 megabytes free hard

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

drive space	drive space
Recommend 1 gigabyte	
with 12 ms	
access time	
24 megabytes RAM	8 megabytes RAM (16 megs recommended)

Software

The following software requirements listed in Table 7 are minimum requirements.

TABLE 7

For your NetWare server	For your client machines
Novell NetWare 3.11	Windows 3.1
Novell Telephony Services, v. 2.2	MS-DOS 5.0
ATT G3 PBX Driver , v. 2.2	Win32S (for IIR Simulator only)
Btrieve 6.15.525	Btrieve ODBC Interface (v. 1.0) for Windows
	Btrieve Client Engine for Windows, v. 6.15

Getting Help

The following IIR User Guide documentation explains how to use the Intelligent Information Router. Chapter 1 describes the basic requirements needed to run IIR successfully, explains how to install the program files for both the server and client components, and discusses the architecture of the system. Chapter 2, "Using the Database Administrator," describes the graphical user interface the Database Administrator uses to set up and modify the fields on the Agent Station. Chapter 3, "Using the Agent Station," discusses the features and operating techniques for the graphical user interface used by the Agents. Chapter 4, "Writing Scripts," provides detailed instructions for operating the script editor and the script simulator window. Chapter 5, "Testing and Implementing your Scripts" explains how to test scripts and how to move them to your production environment. It also introduces the VDN Administrator module. Chapter 6, "Using the Scripting Language," provides comprehensive information for writing scripts. This chapter describes the components of scripts in detail. Appendix A provides reference commands for the scripting language you will use to build the IIR scripts. Appendix B shows samples of typical scripts. Appendix C lists the terms and acronyms used in this manual. Appendix D lists the error messages and the appropriate actions to take for each. From within the program, you can also open the online help topics for quick access to commonly-requested information. Each module has its own self-contained help file. You can open online help from the help menu in any major window for the topics specific to that window.

THIS PAGE BLANK (USPTO)

Understanding Typographical Conventions

The information below will help you find and use the information in the documentation for the Intelligent Information Router. Sequential instructions are numbered steps that must be followed in the order in which they are presented. Filenames appear in this font: `autoexec.bat` (filename) to set it apart from information in the text. Text that you are to type, such as text for scripts, is shown as this: `declare local variable`.

Installing the Program

The installation routine for the Intelligent Information Router allows you to install all or a portion of the program files.

You will need to know a few details about your system to install the files successfully, so please take a moment to review this chapter before you begin to ensure that you have all the information you will need.

1. Read each section carefully before you begin.
2. Write the information required for each step beside the illustrations as you review the chapter to make it readily accessible at the appropriate step.
3. Open Windows (Program Manager or File Manager).
4. Make sure that you have Btrieve installed on the server and the Btrieve ODBC Interface for Windows installed on the client (see Hardware Requirements).
5. Make sure that you are logged on to the NetWare server when you begin installing the program.

Installing Over Existing Files

If you are reinstalling any of the IIR modules over a previous installation, you must follow the steps below before beginning the new installation.

1. Make certain that any existing NetWare Loadable Modules/(NLMs) are not running on the server. All existing NLMs must not be running when you install the NLM. Check this by running "modules" at the NetWare system prompt. Key IIR NLMs include ICRCTL, ICRL, CTNET, CTIPX, CTUTS. To Unload Existing NLMs, quit ICRS CONSOLE and unload ICRS by running "ulicrs" at system console prompts. Also, if you are installing NLMs on a separate NetWare and Telephony servers, you must move three NLMs (`osfasnl.nlm`, `tslib.nlm`, `attpriv.nlm`) from NetWare server to the Telephony server.
2. Back up your existing scripts and databases (both production and simulation) in a directory other than the one to which you are installing now. The current installation overwrites some of these files, though you should receive a warning prompt for this action.

Beginning the Installation Routine

To start the installation,

THIS PAGE BLANK (USPTO)

1. Insert Disk 1 of the set of installation disks into your floppy disk drive.

2. From the File menu, select Run.

3. In the command line, type a: [backslash] setup

where a: is the floppy drive.

4. Press Enter.

Entering Your User Information

As shown by FIG. 6A, The first window to appear is the setup window 500. This window identifies you as the user.

1. Type your first and last names 501.

2. Type the name of your company 502.

Installing the System Modules

As shown by FIG. 6B, in the IIR Setup window 503, you can select which of the IIR module you want to install. You may install the modules in any order, although if you are installing the IIR on a system with no previous IIR installations, it usually works best to install from top to bottom (i.e., begin with NLMS; end with database administrator).

Installing All Modules-General Information

To use the setup window 503 for each module 504, follow the general options below.

1. Click the button of the module 504 you want to install (the order is unimportant).

2. In the setup window, type the information required.

When typing a path and filename, include the drive. For example,

c: [backslash] icrs [backslash] admin [backslash] admin.exe

If you type a path for a directory that does not exist, the IIR installation routine creates the directory for you.

3. Click the Start Install button.

4. Change diskettes when prompted.

5. Click the OK button in the dialog box announcing successful installation.

6. Select a program group for the icon.

7. (Optional) Check the option for Add to Start Up Group to open the module every time you start Microsoft Windows.

THIS PAGE BLANK (USPTO)

8. Click the OK button to return to the IIR Setup window 503.

In the IIR Setup window 503, you can continue installing IIR modules, or you can exit the window.

Notes About the Database Path

When you install the modules, the installation looks for existing database files. If you have existing database files, a dialog box appears and asks if you want to overwrite the old database. Use the following guidelines. If you have backed up the database file, click the Continue button (you can copy your old database file later into the directory if necessary). If you have not backed up the database file, but do not want to overwrite the file, click the Quit button to stop the installation process and return to the IIR Setup window. If you want to overwrite the old database, click the Continue button. When you install more modules within the same installation session, the message will not appear again.

NLM

As shown by FIG. 6C, the NLM window 505 installs the NetWare Loadable Module (NLM) on the server. If you are reinstalling this module over an old one, make certain that the existing IIR NLM module is not running.

Type the following information. For Mapped Drive and Path 506 type the drive and path where IIR will install the files. For NetWare Volume 508 type the server's volume name (for example ASI1). For Switch ID 510 type (pull-down list only) the name of the telephony switch (should display the default switch ID).

Simulator

As shown by FIG. 6D, the simulator window 511 installs the script simulator, designed to run on Win 32S. For Install Path 512 type location to install the Simulator module. For Database Path 514 type location for the IIR to create the database. If this path does not exist, the IIR creates it. If the path and database do exist, the IIR overwrites them. For User Ini Files Path 516 type location for the user initiation files.

VDN Administrator

As shown by FIG. 6E, the VDN Administrator window 517 installs the VDN Administrator for matching scripts to the Vector Directory Number (VDN). For Install Path 518 type location to install the VDN Administrator module. For Server IPX Address 520 type IPX address of your server. The IPX address is defined in the autoexec.ncf file in the system subdirectory of the NetWare server. The line titled ipx internal will provide the first eight characters of the IPX address (network address) followed by the entity within the network or server domain (usually "1" for the server). For example, if the line says "ipx internal net 6125100A" then the address is "6125100A.000000000001." For User Ini Files Path 522 type location for your user initiation files.

Agent Station

As shown by FIG. 6F, the Agent Station Window 523 installs the Agent Station module, which Agents use to modify information in the database. For Install Path 524 type location to install the Agent Station module. For Database Path 526

THIS PAGE BLANK (USPTO)

type location of the production database path on the network server. This path is a combination of the network drive with the NLMs and the subdirectory containing the database. For example, if your network drive is L, and your subdirectory is [backslash] iir, then use the database path L: [backslash] iir [backslash] data.

Database Administrator

As shown by FIG. 6G, the Database Administrator Window 527 installs the Database Administrator module, which Administrators use to set up the Agent Stations. For Install Path 528 type location to install the Database Administrator module. For Database Path 530 type location of the production database path on the network server. This path is a combination of the network drive with the NLMs and the subdirectory containing the database. For example, if your network drive is L, and your subdirectory is [backslash] iir, then use the database path L: [backslash] iir [backslash] data.

When you have finished installing the modules that you selected, you can exit the installation process from the Installation Setup window (see FIG. 6B-Setup Window). Click the Exit button. This program returns you to the Windows Program Manager.

Using the Program for the First Time

The IIR does not use a common interface for all tasks. Ordinarily you simply open the IIR folder and click the icon for the module you want to use. If you have just installed the IIR, however, you will first want to understand where to go from here, and why. You have just installed up to four client applications for the Intelligent Call Router: Database Administrator, Agent Station, Simulator and VDN Administrator.

You can, of course, open any of the applications, but the logical sequence below is for using the IIR the first time:

1. Set up your field labels-You must define and set up the labels and properties of the fields in your customer database. To do this, open the Database Administrator. Chapter 2 discusses this module.
2. Populate the customer database.
3. Write the scripts-The scripts are the heart of the IIR. The tool for writing scripts is the Simulator. Chapter 4 discusses the Simulator window and how to use the scripting editor. If you are unfamiliar with script writing, you can read Chapter 6 for a comprehensive overview and guide to the IIR scripting language.
4. Test your scripts-When you have written a script, you must use the Simulator to test it. Chapter 5 explains this process.
5. Associate the script to a Vector Directory Number (VDN)-When you have tested your script and found it to be successful, you must associate it with the VDN, using the VDN Administrator. The last section of Chapter 5 explains how to associate the VDN to the script and create settings for each script.

THIS PAGE BLANK (USPTO)

6. Modify customer records-This step comes only after the IIR is running successfully and executing scripts. To modify records in the customer database, use the Agent Station module, as discussed in Chapter 3.

Chapter 2-Using the Database Administrator

The Database Administrator allows anyone with administration privileges to set up the database and perform other tasks which globally affect the Intelligent Information Router. As shown by FIG. 7, the Database Administrator has a single window 600, which allows you to set up the fields that appear in the IIR Agent Station window. You can label the fields, designate agent privileges (view and modify) for each of the fields, and define the primary key fields.

Setting Up Fields

You may label up to 15 text fields and 5 numeric fields. For each field, you may also designate agent viewing and modifying privileges.

1. Type the labels (maximum 20 characters) next to the field number.
2. To allow agents to view the field, click the View check box.
3. To allow agents to modify the contents of the field, click the Modify check box.

Notes: Check box with an x () means that the option is enabled. Modify option is void unless you also check the View field. Field 1 601 has special weight. If you allow agents to view and modify this field, agents can add or delete any record. If you allow agents to view but not modify this field, you can check the modify option on other fields. Agents can then modify information in fields other than Field 1. They cannot add or delete records. Text fields may contain spaces (for example, an address field: New York). Numeric fields default to integers.

Defining Primary Key Fields

The Primary Key fields 602 protect against duplicate data. You can choose one of three options in the Primary Key Definition area. None 604 turns off the Primary Key Field designation. Field 1 606 sets the first field as Primary Key. Field 1 and 2 608 sets the first and second fields as Primary Keys.

To select an option,

1. Click the option button for the option you want.
2. Click the Apply button to activate your labels and key definition.

The next chapter explains how to use the Agent Station window that you have just defined.

Chapter 3-Using the Agent Station

As shown by FIG. 8, the Agent Station window 700 is predefined by the Database Administrator (described in the previous chapter). With this window,

THIS PAGE BLANK (USPTO)

an agent can modify the information in the database. To modify the fields shown, you must have modification privileges (see "Setting Up Fields").

Searching the Database

The first four fields contain information that the IIR engine uses to search the database. When it finds a match, it displays all fields for the record.

To search the database,

1. Type the search information in the Search Field(s) 701.
2. Click the Search button 702.

The search results appear in the fields 703 outside/below the Search Fields section. To move forward or backward in the database, click the Prev 704 or Next button 706. These buttons scroll one record at a time.

Using Wild Card Characters

When you search the database, the IIR produces exact matches only, with one exception. If you type "Donald" but have no matching data, IIR continues searching for any wild card characters to the right of the letters you type. So, for example, if the database contains a Donaldson, the IIR also finds that name.

Multiple Characters

If you are unsure of the spelling, use the percent sign as a wild card. For example, if you want to find a name but are not sure if the spelling is Anderson or Andersen, you would type "Anders%n." The IIR search engine finds all records that match the other letters: Anderson, Andersen, Andersan, Andersun, Andersian (the wild card can represent more than one letter). The search engine would display the first available match and you could then use the Prev or Next buttons to scroll back and forth through additional records. Depending upon your ODBC drivers, the search may or may not be case sensitive.

Single Character

You can also search for a single wildcard character by using the underscore character (-). (Insert: (-)). For example, in the case above with the name Anderson, if you type Anders-n, the search would not match the name Andersian, because it has two characters between the letters s and n.

Modifying the Database

Modifying the database includes adding, changing, or deleting information. Because the changes affect the record in the database, you must have privileges assigned by the Database Administrator.

Adding a Record

To add a record you must have modification privileges to Field 1.

1. Type the information in the fields 703 below the Search Fields 701.

THIS PAGE BLANK (USPTO)

2. Click the Add button 708 to create a new record.

Modifying a Record

To modify a record, you must have modification privileges to the field you want to modify.

1. Search the database for the record you want to modify, if it is not already displayed.

2. Click the Update button 710 to change the information.

Deleting a Record

To delete a record, you must have modification privileges to Field 1.

1. Search the database for the record you want to modify, if it is not already displayed.

2. Click the Delete button 712.

The record erases the record from the database and is not recoverable.

Chapter 4-Writing Script

Scripts are powerful tools that enable you to write a set of commands to automate for running repetitive tasks. The Intelligent Information Router engine reads and executes the commands, which you write in a special scripting language. The scripts are made even more powerful by allowing you to use conditional logic.

This chapter introduces you to the scripting process, including creating, modifying, and printing with the IIR script editor.

Using the Simulator Window

As shown by FIG. 9A, the Simulator window 800 has two sections: script section 801 and output section 803. The script section in the upper portion of the window displays the script windows. Each window serves as a script editor for writing and modifying your scripts.

Use the output section 803 in the bottom of the window to view the results of testing your script commands. When you open the window, this section also contains initialization information for related files. If you do not see this information, the simulator is not properly initialized.

You may open multiple edit windows and organize them in various arrangements (tile, cascade, etc.). The simulator window 800 shown in FIG. 9A has two scripts open. By default, the window opens with a blank page in the script window. When you save this script, the name of the script appears in the title bar of the script window.

Changing the Appearance of the Simulator Window

THIS PAGE BLANK (USPTO)

You can change the appearance of the Simulator window 800 to allow more workspace. From the View menu 802, click Toolbar and/or Status Bar to hide or unhide these objects. From the Windows menu 804, select Cascade to stagger your script windows. From the Windows menu 804, select Tile to see the scripts in rows. From the Windows menu 804, select Arrange Icons to align minimized (iconized) script windows. Move the output window. This window is dockable, meaning that if you move it to another border, it attaches itself to the border. You can also use it as standalone. To move the output window, grab it with your mouse and drag to another location. If you close the output window, you cannot reopen it until you exit the Simulator window and reopen.

Writing a Script

The IIR allows you to write as many scripts as you need. The IIR Script Editor is a simple ASCII text editor with a few enhanced options. With the IIR script editor, you can use formatting options (including fonts) and you can copy from other scripts, using standard Windows-based copy-and-paste techniques.

Opening a Script

When you open the Simulator application, a blank window appears by default. You can use this window to write a new script, open additional new windows, or open existing scripts.

New Script

If no blank windows are available, or to open additional blank windows,

1. From the File menu, select New.
2. Proceed to the section on page 19, "Formatting the Script."

Existing Script

To open an existing script,

1. From the File menu, select Open.
2. In the Open dialog box, select a script (scripts have the file extension .icr). This dialog allows you to search your directories for scripts. When you install the IIR Simulator, it creates a subdirectory named userscr for storing your scripts. If you want to have this dialog box open to this subdirectory, follow these steps. In Windows program manager, click to select the IIR Simulator icon. From the File menu, select properties. In the "working directory" field, enter the full path of the userscr directory. For example: c:\[backslash] iir [backslash] sim [backslash] userscr. Click the OK button.

3. Click the OK button.

4. Proceed to the section, "Formatting the Script."

Formatting the Script

The general procedure for editing a script includes the steps below:

THIS PAGE BLANK (USPTO)

1. Place your cursor in a script window.
2. Begin typing. For information on the IIR scripting language, please refer to Appendix B.
3. From the File menu, select Save or Save As to save your script. The IIR uses the standard Windows Save dialog.

The formatting options are available from the Settings menu. Select Tab Stops to set the number of spaces to use for tabs. You may select from 1 to 16 spaces for tabs. The tab settings affect only the active script. Select Editor Font to select a font for the script editor (see "Choosing Fonts" on page 19 for information). The font setting affects all script windows. Select Word Wrap to wrap text from one line to the next without a line break. The Word Wrap option affects only the active script.

Copying from Other Scripts

The IIR script editor allows you to copy from other scripts by using Windows techniques for copy-and-paste. The script editor does not support drag-and-drop copying. Please refer to your Windows documentation if you need help with these techniques.

Choosing Fonts

Font choices apply to all scripts. You cannot maintain scripts with different fonts. IIR allows you to select True Type fonts for both screen and printer, although these fonts may produce undesirable results. Most True Type fonts are proportional, which means that each character uses a different amount of space. For example, the letter i uses less space than the letter w. In contrast, nonproportional fonts such as the Courier typeface are proportional. All these letters take up exactly the same space.

For printing, proportional fonts are considered easier to read. For formatting scripts, however, you cannot always vertically line up individual characters or lines of text if you use proportional fonts. If spacing is important to you, use a nonproportional font such as Courier, Terminal, MS Sans Serif, or MS Serif.

Also, you can edit any IIR script in any ASCII text editor, but not all text editors can use proportional fonts. If you open a script in one of these editors, your spaces and alignments will change and font appearance may be unpredictable.

Using the Find Dialog

To find a string of characters,

1. Place your cursor in the script you want to search.
2. From the Edit menu 808, select Find.
3. As shown by FIG. 9B, in the Find What field 809, type the string.

THIS PAGE BLANK (USPTO)

4. (Optional) Check the Match Case option to find only occurrences with exact match of uppercase and lowercase letters.

5. Check the direction for your search, Up (this point to beginning of script) or Down (this point to end).

6. Click the Find Next button to go to the string.

The dialog box does not close, allowing you to continue searching through the script text.

Using the Find and Replace Dialog

To find and replace a string of characters:

1. Place your cursor in the script you want to search.

2. From the Edit menu 808, select Find and Replace.

3. As shown by FIG. 9C, in the Find What field 811, type the string that you want to replace.

4. In the Replace With field 813, type the new string.

5. (Optional) Check the Match Case to find only occurrences with exact match of uppercase and lowercase letters. The editor replaces these strings with the exact case of the string in the Replace With field.

6. Click the Find Next button to go to the string.

7. When the string is found, click the Replace button to replace the string and find the next occurrence.

Or, click the Replace All button to replace all matching strings.

8. Click the Cancel button to close the window.

The dialog box does not close, allowing you to continue searching through the script text.

Saving Your Script

You can save your changes at any time. When you close the script or exit the IIR Simulator with unsaved changes, a dialog message prompts you to save your changes.

To save anytime, choose one option. From the File menu 806, select Save. If you are saving an existing script, this dialog saves changes to the existing file name. From the File menu 806, select Save As. If you are saving an existing script, this dialog prompts you for a new filename. It saves changes to the new filename, leaving the old file unaffected.

To save changes when you exit the IIR Simulator,

THIS PAGE BLANK (USPTO)

1. From the File menu 806, select Exit.
2. In the Save dialog message, choose click one button. Yes-Saves the changes to the filename shown in the dialog. No-Closes the application without saving any changes. Cancel>Returns to the program. Changes in your script are still intact, but the script is still unsaved.

Printing a Script

As shown by FIG. 9D, the IIR editor allows you to print your screen to paper or a file (ASCII). Your dialog window 820 may differ slightly from the one in FIG. 9D, depending upon your printer driver. Please refer to your Windows documentation if you need help with using the Print dialog window.

Previewing a Script

Before you print, you can see your script as it would print.

1. From the File menu 806, select Print > Preview.
2. Use the command buttons at the top of the Preview window to change the view.
3. Click the Close button to return to the editor.

Setting up the Printer

The IIR uses the Windows print setup dialog.

Selecting a Printer Font

By default, the IIR uses the System (non-True Type, nonproportional) font. You can change this font. Please refer to "Choosing Fonts" for information about proportional versus nonproportional fonts and how they affect your scripts.

To select a printer font,

1. From the Settings menu 822, select Printer Font.
2. Select Change to open the Font dialog box. Select Same as Display to use the same font that you use in the editor (no dialog box).

The next chapter provides more information on using the Simulator window.

Chapter 5-Testing and Implementing Your Scripts

The following stages explain the basic processes of creating and using scripts. Create the script, using the script editor in the IIR Simulator window. Test your scripts with simulated data and view their output in the IIR Simulator window. Implement the scripts with the VDN Administrator. This module sets up the scripts to work on your development platform. This chapter explains how to test and implement the scripts.

Testing Scripts

THIS PAGE BLANK (USPTO)

The IIR Simulator opens with a blank page in the screen editor. As shown by FIG. 9A, you can use the editor to write a script on this blank page or open an existing script. Generally, your scripts fall into one of two categories: telephony (involving data received from an incoming call or data supplied by the caller dialing more digits) or non-telephony (scripts which do not test telephony calls). The non-telephony calls probably represent a very small percentage of your scripts.

Testing Non-Telephony Scripts

To test a script,

1. Open the script if it is not already open.
2. Click the title bar of the script to make it active. A script must be active (on top, if you have more than one screen showing) for you to test it.
3. (Optional) From the Settings menu, select Hide While Running to hide the entire window while you are running the Simulator.
4. From the File menu, select Run Direct.

Or, click the Simulate icon on the toolbar.

Testing Telephony Scripts

To test a script,

1. Open the script (see "Opening a Script" for help) if necessary.
2. Click the title bar of the script to make it active.
3. (Optional) From the Settings menu, select Hide While Running to hide the entire window while you are running the Simulator.
4. From the File menu, select Simulate.

Or, click the Simulate icon on the toolbar.

Using the Telephony Handle Simulator

As shown by FIG. 10A, a new window 910 appears if you are simulating a telephony script. This window allows you to enter the information here to simulate the information that the route request would ordinarily provide for a live call.

To use this window

1. Enter the information you want to simulate. See Table 8 below.
2. Click the OK button.

The results of the test or simulation appear in the output window.

TABLE 8	
Fields	Description

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

VDN	Enter the VDN that would normally handle calls for this script. If you do not use VDN in your script, you do not need to enter anything in this field.
CallingDevice	Enter the calling device that represents a call that would normally route through your script. If you do not use CallingDevice in your script, you do not need to populate this field.
Trunk	Enter a trunk group that represents a call that would normally route through this script. If you do not use Trunk in your script, you do not need to populate this field.
UUI	Enter User-To-User Information (UUI) that represents a call that would normally route through this script. If you do not use UUI in your script, you do not need to populate this field.
UserEnteredCode Digits	Enter UserEnteredDigits that represents a call that would normally route through this script. If you do not use UserEnteredDigits in your script, you do not need to populate this field.
UserEnteredCode Type	Enter UserEnteredType that represents a call that would normally route through this script. If you do not use UserEnteredType in your script, you do not need to populate this field.
UserEnteredCode Indicator	Enter UserEnteredIndicator that represents a call that would normally route through this script. If you do not use UserEnteredIndicator in your script, you do not need to populate this field.
UserEnteredCode CollectionVDN	Enter UserEnteredVDN that represents a call that would normally route through this script. If you do not use UserEnteredVDN in your script, you do not need to populate this field.
LookAheadInfo Type	Enter LookaheadType that represents a call that would normally route through this script. If you do not use

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

LookAheadInfo Priority	LookaheadType in your script, you do not need to populate this field. Enter LookaheadPriority that represents a call that would normally route through this script. If you do not use Lookahead Priority in your script, you do not need to populate this field.
LookAheadInfo Hours	Enter LookaheadHour that represents a call that would normally route through this script. If you do not use Lookahead Hour in your script, you do not need to populate this field.
LookAheadInfo Minutes	Enter LookaheadMinutes that represents a call that would normally route through this script. If you do not use Lookahead Hour in your script, you do not need to populate this field.
LookAheadInfo Seconds	Enter LookaheadSeconds that represents a call that would normally route through this script. If you do not use Lookahead Seconds in your script, you do not need to populate this field.
LookAheadInfo SourceVDN	EnterSourceVDN that represents a call that would normally route through this script. If you do not use Lookahead SourceVDN in your script, you do not need to populate this field.

When you have entered the information required, you can begin the test. Click the OK button 912 to begin the simulation. Click the Cancel button 914 to end the Simulation.

Viewing the Results

As shown by FIG. 10B, if you have print or println commands in your script, the results of your test appear in the output window 920 at the bottom of the screen.

Changing the Position

You can move the output window to any position on the screen, including leaving it as standalone or docking it to the top or sides of the window, but

THIS PAGE IS BLANK (USPTO)

not resize it.

To move the window,

1. Click the window in the gray border area.
2. Drag to the new location.

To clear the output window, select Clear Output from the Edit menu.

Making the Script Available

When you have tested a script and viewed the successful output, you are ready to make the script available to your production environment.

To copy the script to another location,

1. Click the title bar of the script to make it active.
2. In the File menu, select Save As.
3. Save the script to your production subdirectory on the NetWare server usually [backslash] userscr.

You can also use Windows File Manager to copy the script to the appropriate subdirectory. When the script is in place, it is ready for you to associate with a Vector Directory Number (VDN), using the VDN Administrator.

Using the VDN Administrator

As shown by FIG. 10C, the VDN Administrator matches the scripts to the VDN and starts and stops the scripts on the production system. When you open the VDN Administrator window 930, a list box 932 displays all scripts that are set up for the system. You can click the column title (VDN or Script) to sort the list on that column.

Adding a Script

To add a script,

1. Click the New button 936 to open the VDN Properties window 938 shown by FIG. 10D.
2. Enter a VDN to associate with the script.
3. Enter the script filename (use the Browse button to find it).
4. Check one or both options. Set Default Route Session 940 to set all unassigned VDNs to run a designated script. Set Start Script Route Session 942 to assign a VDN to a script and sets the script to start immediately. You may set only one script for one VDN, but you may set more than one VDN for a single script.

Starting and Stopping a Script

THIS PAGE BLANK (USPTO)

Returning to FIG. 10C, if a script is not started or is inactive, the check box is empty and the script name appears as light gray. You may start a script with any of three methods. Click the Start button 943. Click the check box next to the VDN in the VDN Administrator window. Click the check box for Start Script Route Session in the VDN Properties window.

Refreshing the VDN Settings

The Refresh feature is helpful for determining which scripts are currently active or "started." For example, if a script is accidentally deleted or stopped, this window reflects the status when refreshed.

Using Refresh Command

To force the VDN Administrator to refresh the settings at any time, click the Refresh button 944. You can also set the screen to refresh automatically as shown in the next section.

Setting the Refresh Time Rate

As shown by FIG. 10E, to set the automatic refresh rate,

1. Enter the refresh time using seconds.
2. Click the Start Timer button 946.

The default is ten (10) seconds. You may set any value from 10 to 64. To set the refresh to never refresh automatically, click the Cancel button 948.

Deleting a Script

Returning again to FIG. 10C, the Delete command in the VDN Administrator window does not remove the script from the directory. Instead, it removes the script from the window only.

1. Select the script you want to remove.
2. Click the Delete button 950 below the script list box.

Modifying the VDN Settings

Use the VDN Properties window to modify settings also.

1. Select a script in the list box.
2. Click the Modify button 952 to open the VDN Properties window 938.
3. Change the settings (see "Adding a Script" for further information).
4. Click the OK button to save your changes and close the window.

Chapters 5 and 6 have explained the overall concepts of the IIR Simulator window. The next chapter provides a detailed look at the IIR scripting language.

THIS PAGE BLANK (USPTO)

Chapter 6-Using the Scripting Language

The Intelligent Information Router (IIR) system is designed to route calls in a call center environment. The IIR is more intelligent and flexible than the existing ACD/PBX equipment, using the IIR Script as the mechanism through the system obtains the advanced intelligence and flexibility.

To develop the IIR scripts, you can use the IIR Simulator, a primary component of the IIR. The IIR Simulator provides an interactive script development environment for testing and debugging of scripts without leaving the tool. If you have not already become familiar with the IIR Simulator tool, please read the sections on "Writing Scripts" (beginning on page 17) and "Testing and Implementing your Scripts" (page 23) before continuing.

This chapter provides development-related details on the IIR script language, including standards, pragmas, and syntax.

Understanding the IIR Environment

It is important to understand how the IIR scripts integrate with other components in the call center. The IIR is an intelligent adjunct router when viewed from the switch perspective. The AT&T switch views the IIR as one of possibly several adjunct processors that will provide call routing direction back to the switch when requested.

Although the PBX/ACD can make rudimentary decisions about routing of calls through vector processing commands, it makes global routing decisions rather than customer-oriented routing decisions. For example, the PBX/ACD can determine routing based upon the number of agents available for a particular queue or time of day (for after-hours handling). The routing capability stops there, however, as the PBX/ACD does not have access to customer specific data nor the operations available to process such data in the vectoring commands. Although this type of routing was an advanced feature a few years ago, today's call centers are demanding more customer-oriented routing capabilities which is where the IIR takes over.

By integrating the IIR to the PBX/ACD as an adjunct routing controller, an option available through the vectoring capabilities on the switch, the IIR easily performs customer-specific routing. Through sophisticated logic control combined with access to customer-specific databases, the IIR increases productivity and service by allowing customer-specific routing.

Customer-specific routing can combine information such as ANI, customer-entered digits, or the VDN from which the caller is being routed to make determinations about the routing destination of the caller. For example, when the IIR receives the adjunct route request, it also receives the TSAPI route request message containing all pertinent information about the caller including ANI, originating VDN (DNIS), user-entered digits (DTMF collected), and many other fields. The IIR then uses this information to provide a database lookup and an eventual route based upon the logic in the IIR script.

The service and efficiency possibilities become almost limitless with this intelligent adjunct router. Consider the following scenario where the routing of the caller is determined by multiple factors in a specified order of priority.

THIS PAGE BLANK (USPTO)

A call center manager needs to provide three levels of service for the bank card customers. All bank card customers have been classified into three membership categories including Platinum, Gold, and Standard. The customer classification, along with the customers home phone, business phone, account number, current balance, account limit, and other information are stored in the IIR customer database. The call center manager designs multiple service queues for the different customer classification to provide the shortest waits along with the most experienced Customer Service Representatives (CSRs) to the Platinum, while the standard customers receive the longest queue times and least experienced representatives.

The switch sends all service calls to the IIR, The IIR then performs a lookup on the ANI information. If it finds a match, it determines the customer classification and returns the route for the appropriate agent queue. If it does not find a match, the IIR performs a RouteMore command that sends the call back to the switch for collection of account number digits. Then, based upon these digits, the IIR performs a lookup, matches the customer classification, and routes the call to the correct agent queue. Through this very simple case, the call center has transparently provided differentiated service without having to manage, publish, and maintain updates on different marketing numbers for different classes of service.

At this point, the call center administrator might decide to provide continuity to its Platinum members by always queuing them to a specific customer service agent, if available, while a customer related problem maintains an open state. The IIR can then match the service, determine Platinum, and then route the call to a specific agent, based upon dynamic updates made by the customer service agents. The IIR can determine not only if a specific agent is logged in, but also determine whether the agent is currently on an active call. By using combinations of priority and agent skills, the IIR could queue Platinum callers to a specific agent for preferential treatment.

The routing possibilities are extraordinary with the use of the IIR adjunct. The remainder of this chapter will explain how to develop an IIR script instead of the logical application. Scripting constructs, control flow, program structure, and script language reference will provide you with the knowledge to empower your call center.

Integrating IIR with the AT&T Switch

The section above discussed at a high level the integration of the IIR and some possible applications. Before attempting the language constructs, you will need to examine the functional integration of the IIR with the AT&T switch.

The first integration issue to consider is the component which actually controls whether the IIR is requested to determine a route: the PBX/ACD. The mechanism on the PBX/ACD which invokes the IIR is a combination of the Vector Directory Number (VDN) and call vectoring. Call vectoring is a limited "scripting" language on the switch which provides sequential handling of calls. A standard part of the call vectoring language is the adjunct route request step.

The adjunct route request step requires that the adjunct connect to the PBX/ACD via the ASAI link. In this case, a system running telephony services for the AT&T DEFINITY G3 connects to the switch and serves as the "information

THIS PAGE BLANK (USPTO)

link" between the IIR and the switch. When the adjunct route request is encountered in the call vectoring sequence, the switch sends a Route Request Service from the switch to the IIR via the telephony server. This route request contains detailed information about the call that the IIR uses to route the call (which includes ANI, originating VDN, trunk, user-entered data [DTMF collected digits], and other information).

The IIR, a client to the telephony server, receives the route request service event and associates the originating VDN with one of the IIR scripts. All of the information in the Route Request Service request is available within the IIR script to use in determining the appropriate route. The IIR also has access to a customer database using one or several of the route request fields as a key to the database (i.e., it can match ANI against the phone number in the database to determine a record match).

Once the IIR has determined where to route the call, it issues a Route Select Service message back to the PBX/ACD via the telephony server. The Route Select Service message contains a destRoute field that the IIR fills in with the destination extension number (can represent VDN, agent login-id, etc.). The Route Select Service message can also be sent back with other fields used for application to application communications.

The general concept is fairly simple. The AT&T switch requests a route to be performed through the adjunct route request step in call vectoring. This sends a Route Request Service message to the IIR. The IIR determines which IIR script to execute based upon the origin VDN and executes this script. The script interacts with the customer database, applies logic to the information, and determines a route destination. This information is placed into the Route Select Service message which returns to the call vector awaiting routing direction.

The logic within the IIR script varies with each script, but the same basic process exists in each script. The following sections detail the actual functions used to perform these operations, along with details on all logic capabilities available in the IIR script language.

Using Operators and Expressions

This section explains operators and expressions of the IIR script language. Operators include arithmetic, logical (comparison), assignment operators, and others. Expressions include variable declarations, assignment expressions, and general expressions. The sections below explain how operators and expressions interact.

Variables (Declaration, Usage)

Variables fall into two classifications in the IIR scripting language: local and global. You must define all variables before referencing them in the script. Normally, you declare the variables near the top of a script, although you can declare them just before the section where they are used.

Local variables require the keyword local before the name of the variable. All local variables are variant in nature: they can contain either numeric (integer or float) or string data. The information associated with a variable depends upon the type of data assigned to the variable. Normally, the assignment is to a numeric or string valued returned from one of the IIR functions.

THIS PAGE BLANK (USPTO)

Local variables are created for each script execution and only exist throughout the execution of the script. Future invocations of the same script for a different call will reinitialize variables to default values. Local variables can be declared one per line, or can be combined on a single line within the script as shown by Example 5.

Example 5

```
local      varOne
local      varTwo
local      varThree
local      varOne, varTwo, varThree
```

Variable names can consist of up to 32 characters, and can have a combination of numeric and alpha characters and mixed case. Good programming practice dictates comments for each variable or set of variables. Comments can exist on the same line as the declaration, and each is prefaced with a '#' (pound) sign. The value of a local variable cannot be referenced until it has been assigned a value within the script as shown by Example 6.

Example 6

```
local      varOne      #This is the first variable
              *          in the script
local      varTwo      #This is the second
              *          variable, used for string
              *          manipulation
local      varThree    #This is the third
              *          variable . . .
```

Global variables are IIR specific variables that can be referenced or set in the script. These variables normally pass system information into the script (such as the script name, script execution id, or logging level for the particular script).

The global variables are not global between scripts, but instead apply to the instance of the script being executed. In addition, no global variables other than those defined by the system can be declared. In other words, the global definition only provides access to pre-existing system global variables and is not a declaration of a new global variable. Global variables are defined in the script by prefacing the variable name with the keyword global as shown by Example 7.

Example 7

```
global      g-nICRLLogLevel      # Global variable
              *                    to control logging
              *                    level from this
              *                    script
```

THIS PAGE BLANK (USE TO)

```

Pat. No. 5870464, *
global      $ 0, $ 1      # $ 0 is the name
                *          of the script
                *          being executed
                *          # $ 1 is the
                *          telephony event
                *          handle used to
                *          obtain the call
                *          information

```

nNOTE: Global variables '\$ 0' and '\$ 1' are necessary for obtaining the route request information for all telephony functions. These global variables should be, as a practice, exposed in all scripts by declaring them as shown above. -

Simulating Constants with Variables .

This scripting language does not provide true constants, although good programming practice defines at the top of the script the local variables that would normally be set as constants for ease in maintenance.

The following examples of Example 8 show variables declared and then assigned to be used as constants throughout the application. Note that the variables have been declared in all capitals, which is not necessary but allows for differentiation throughout the script.

Example 8

```

local      RET-OK          # Valid return from all
                *          but handle allocations
                *          of 0
local      DEFAULT-RTE    # Default route to send
                *          caller to
local      AFTERHR-RTE    # After hours default
                *          route

```

```

RET-OK = 0
DEFAULT-RTE = 2999
AFTERHR-RTE = 3999

```

By defining these values at the top of the script, you can easily modify these values in a single location, even if you reference them many times in the script. This practice is particularly useful in setting up default routes that might change on an infrequent basis.

Arithmetic Operators

The binary arithmetic operators are (+ , - , * , /). You can apply these operators to any numeric variables in the script, basing the result upon whether the arguments are integers or float. If either variable being operated on is float, the result is a float. If both variables are integer in type, the

THIS PAGE BLANK (USPTO)

result is integer. You may combine integer and float variables in arithmetic operations without casting.

The following Example 9 shows examples of code and the result:

```
Example 9
local      numOne, numTwo, result      # declare
          *                             variables for math
          *                             ops

numOne = 4.8
numTwo = 2
result = numOne/numTwo < result = 2.4 >
result = numTwo * numTwo < result = 4 >
result = numOne - (numTwo * numTwo) < result = 0.8 >
```

Relational and Logical Operators

The following relational operators in Table 9 can compare numeric as well as string data.

TABLE 9

```
" == " - comparison of equal
" < > " - comparison of does not equal
" < " - comparison of less than
" > " - comparison of greater than
" < = " - comparison of less than or equal to
" > = " - comparison of greater than or equal to
```

The logical operators for forming expressions are shown by Table 10:

TABLE 10

"Not" "And" "Or"

Some legal combinations of the above in an 'if' statement are shown by Example 10:

Example 10

```
# If varOne is less than varTwo and varOne is not
zero

if ( (varOne < var Two) And (varOne < > 0))
```

THIS PAGE BLANK (USPTO)

then . . .

if ((varOne < varTwo) And (Not varOne)) then . . .

Assignment Operators and Expressions

The only valid assignment operator is the " = " (single equal sign). This assignment operator can be used to assign numeric or string values to any variable. For example, the following assignments in Example 11 are all valid:

Example 11

```
local variantVar
variantVar = 2
variantVar = 3.0
variantVar = "Eric"
```

The variable contains the last value assigned to it. The expressions to the right of the " = " assignment operator can be complex and can also evaluate to a binary value. In this scripting language, binary values are stored as integers with '1' as true, '0' as false. See Example 12.

Example 12

```
local varOne, varTwo, result
varOne = 10
varTwo = 5
result = ((varOne/varTwo) == 2)
```

```
println "Result is = " + result
```

```
< would print
"Result is =
1" >
```

The following Example 13 shows the precedence order from top (highest) to bottom (lowest) of the expression operators.

Example 13

```
( ) (parenthesis always override other precedence rules)
* / (multiplication and division)
+ - (addition and subtraction)
< < = > > = (relational operator)
```

THIS PAGE BLANK (USPTO)

= = < > (equality checking)

And Or Not (logical operators)

= (assignment operator)

All operators on the same line are evaluated from left to right. All operations on a line higher than the line on which an operator is found will be evaluated before the operator in question. If you have any question about the precedence order of evaluation, use "()" (parenthesis) to assure it is evaluated in the intended order. The following Example 14 shows how an expression would be evaluated based upon the above rules of order.

Example 14

```
local numOne, numTwo, result
numOne = 3; numTwo = 2
result = numOne + numTwo * numOne < result = 9 >
```

Why is the result 9? The order of evaluation shows that all multiplication and division will be performed before addition and subtraction. Thus 'numTwo * numOne' is evaluated first, resulting in 6, and then added to numOne last. So the default evaluation, if containing parenthesis, would look like the following Example 15.

Example 15

```
result = numOne + (numTwo * numOne)
```

Always use parenthesis whenever doubt exists about the order of evaluation.

Script Output and String Operators

The string operators in the IIR scripting language are very easy to use. Strings can be combined using the ' + ' operator. In addition, non-string variables will automatically be converted to their string equivalents by adding them to an existing string as shown by Example 16.

Example 16

```
local numVar, strVar, strVar2
numVar = 3
strVar = "This is a string, and a number"
strVar2 = strVar + numVar
```

```
< result would be
"This is a string,
and a number 3" >
```

THIS PAGE BLANK (USPTO)

Multiple strings and variables can be concatenated on the same line. In addition, the comparison operators defined above in the relational operators section can be used to determine the equality of two strings.

Strings can be output in both the simulation and production modes. The output in the simulation mode is directed to the small output window in the lower portion of the simulator window. The output appears on the production system if the output is left intact in the script. The location of the output will be the ICRS Console session.

The commands for producing output to the debug window and ICRS console are `print` and `println`. Both commands take string arguments or expressions that evaluate to strings. The `print` statement does not go to a new line after printing the string, but the `println` statement does. The `print` and `println` functions are extremely valuable for debugging in the simulator, but may result in cumbersome output to the screen on the production server. For this reason, consider commenting out all `print/println` statements in the script after you are convinced of its correct operation.

For the production server, the `print/println` can be a useful tool if common debugging statements are removed by only outputting those errors or events considered to be critical in nature. This method provides a real-time monitor of critical errors that can easily be monitored for activity.

One last form of output is via the `logExpr` function. This function works exactly like the `println` function with one difference: the destination of the output. The `logExpr` function's output is destined for the LOGn files in the [backslash] LOG subdirectory off of the simulator directory on the development system and off of the [backslash] ICRS directory on the production system.

Comments

Comments exist on any line of the script. The pound sign ('#') precedes all comments and defines the remainder of the line as comments. The '#' can be placed at the end of a valid line of logic or declarations, such as shown above in the variable declaration section. Any information following the comment declaration symbol serves as comment only on that line.

The following lines of Example 17 are all valid examples of comments.

Example 17

```
# -----
# Script Name: < your script >
#
# Description: < script description >
# -----
```

```
global $ 0, $ 1      # $ 0 is the script name, $ 1
                    is the call identifier
```

```
local      hTelHandle      # To hold
*          *               telephony handle
```

LANK (USPTO)


```
hTelHandle = ICRLAtoi($ 1)      # Convert the
                                telephony handle
                                to numeric
```

In this example, notice that some of the comments are placed at the beginning of the line, while others follow valid scripting components such as variable declarations.

Control Flow

Statements and Blocks

Statements consist of a single logical expression or assignment, such as Example 18 below:

```
Example 18
varOne = varTwo * 3
println "VarOne now equals " + varOne
```

Multiple expressions can exist on the same line if separated by a semicolon separator(;). The separators provide a means of putting multiple short statements on a single line, as shown in the following Example 19:

Example 19

```
varOne = 3; varTwo = 6; varThree = 14
```

Except for this case, lines in the script do not need to terminate with a terminating symbol. Blocks of statements fall sequentially in the scripting language into groups dictated by the beginning and end of constructs. No specific block begin/end symbol exists, but each construct (such as the if-else-endif) has self-defining block definitions.

If-Else-Endif

The if-else-endif construct expressed decisions. The formal syntax is as shown below in Table 11:

TABLE 11

```
if (expression) then
```

```
< statement block 1 >
```

```
else
```

THIS PAGE BLANK (USPTO)

< statement block 2 >

endif

where the else portion of the statement is optional. The expression is evaluated and if true, < statement block 1 > is executed. If false, and the else portion is included, < statement block 2 > is executed instead.

The < statement block n > may contain any combination of statements including more if-else-endif constructs. The endif portion of the statement must always be specified to terminate the block of statements being executed for the given expression.

The following Example 20 illustrates a three-way decision that would require an if-else-endif construct embedded within the else statement block.

Example 20

if ((tmpVal < 6) And (tmpVal < > 0)) then

< statement block 1 >

else

if (tmpVal == 0) then # tmpVal is 0

< statement block 2 >

else # tmpVal > = 6 and not zero

< statement block 3 >

endif

endif

Deeply nested if-else-endif statements can be cumbersome. If the test is simply to detect one of multiple values, the case statement may be a better operator as described below.

HIS PAGE BLANK (USPTO)

Select Statement

The select statement shown by Table 12 tests whether an expression matches one of a number of constant integer or string values, and branches accordingly. Select Case expression

```
Case < value >
< statement block 1 >
End Case
Case < value2 >
< statement block 2 >
End Case
.
.
.
Case Else
< Statement block n >
End Case
```

End Select

Each case is labeled by only one integer or string value constant. If a case matches the expression value, execution starts at that case and continues to the End Case statement. Each case value must be different, while the case labeled Else is executed if none of the other cases are satisfied. Comparison values must either be integer values or strings (enclosed in double quotes) and cannot be mixed within the same select statement. The Else case is optional. If this case is not present, and if none of the cases match, no action takes place.

Loops: For-Next and Do-Loop Until

Both the For-Next and Do-Loop Until constructs provide a means of looping through multiple iterations of a statement block. The For-Next construct is preferable when there is a known number of iterations for the loop, while the Do-Loop Until construct is preferable when looping until a specific condition is met.

The syntax of the For-Next construct is shown by Table 13:

```
TABLE 13
ForCountVariable = StartToEndStepIncrement
< statement block >
Next
```

In this syntax, Start and End indicate the beginning and ending increment value (for example, to loop from 1 to 20, Start would be 1 while End would be 20). The Increment value represents the step to take between iterations, normally 1.

THIS PAGE BLANK (USPTO)

The Example 21 below shows a For-Next statement followed by an increment other than 1. This loop would iterate 5 times even though the start and end are 1 and 21.

Example 21

```
For CountVariable = 1 To 21 Step 4
```

```
< statement block >
```

```
Next
```

The other looping construct is the Do-Loop Until construct. As stated above, the Do-Loop Until construct is better suited to those conditions where the loop will terminate based upon a condition rather than a set number of iterations.

The Do-Loop Until syntax is shown by Table 14 as follows:

TABLE 14

```
Do  
< statement block >  
Loop Until (Expression)
```

The expression defined for this construct should evaluate to a Boolean expression, although it evaluates to true if the value is integer and non-zero.

Program Structure

At this point, you should be familiar with the basic language constructs and capabilities. This chapter combines this information with the functionality specific to an IIR script. More specifically, the sequence for a "standard" IIR script could be as follows:

1. Receive a route request.
2. Obtain customer specific information from the route request.
3. Access a customer database using one or several of the route request fields as a key.
4. Determine the correct route.
5. Send back the route select event.

This sequence represents a "standard" IIR script, although all IIR scripts may not follow this exact flow. Also, the actual logic varies for determining the appropriate route, depending on elements in the customer record and the specific routing task implemented.

The next section groups and describes the functions available for handling telephony, database, and miscellaneous tasks.

THIS PAGE BLANK (USPTO)

There are basically three logical groupings for the IIR functions. Telephony Event are those allowing read and write access to the Route Request/Select TSAPI events. Database Access are those functions providing access to the customer database. Miscellaneous Operation are those functions supporting such features as date, time, and string manipulation. Detailed information for each function is included in Section 5, Command Reference. Each function defined in the command reference includes a detailed description of the function along with examples of use for that function.

All of the telephony functions require that a "telephony handle" be passed to the call as the first parameter. This "telephony handle" is obtained at the beginning of the script using the global variable '\$ 1'. The global variable '\$ 1' represents the unique call identifier associated with this call and this instance of the running script. To obtain the telephony handle for this call, the '\$ 1' global variable must be passed into the ICRLAtoi function to convert it to an integer handle. This operation needs to be done only once at the beginning of an IIR script to provide access to all of the telephony event information.

```

Example 22
global      $ 1          # $ 1 is the call
              *          identifier text string
local      hTelHandle    # Telephony handle for
              *          events
local      nRet          # Return value from
              *          calls

```

```
hTelHandle = ICRLAtoi($ 1)
.
.

SetRouteSelected (hTelHandle, "3333")           # Route to extension 3333
SetPriorityCal ( hTelHandle, 0)                  # Set priority to off
nRet = RouteFinal (hTelHandl)                   # return Route Select message
if ( nRet < > 0) then
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
println "error on Route Final in
" + \$ 0 + "of " + nRet

endif

The telephony handle is valid from the point in the script that this call is made to the point where a RouteFinal or RouteUnknown is called. These functions inherently free the telephone handle in the system when they route the call. Any references to the telephone handle after invoking one of these functions will result in an error being returned.

In the above example, there are two Set fields which must be defined before performing the RouteFinal (mandatory components of the Route Final Request message). These two fields are priority and routeDest which are set using SetPriorityCall and SetRouteSelected respectively. The route final request will fail if these two fields have not been set before the call to RouteFinal.

Appendix C lists all telephony functions, including function types GetRteReq, SetRteSelect, and GetAgentState. Those functions with type GetRteReq access information from the Route Request message which initiates the scripts execution. Functions having type SetRteSelect set fields in the Route Select message which is sent back to the ACD/PBX with routing instructions. Finally, GetAgentState functions obtain state information on specific ACD agents. For detailed information on a specific telephony function, look at the detailed entry for that function.

Database Access Functions

The general classification of database functions includes all functions used to access customer information from the server based database. All functions that access the database require a database handle. This database handle defines the current record location in the database.

The database handle needs only to be created once at the beginning of the IIR script using the CreateDBHandle() function, and must be released before exiting the script using the DestroyDBHandle() function. The following script segment in Example 23 shows the calls necessary to allocate the database handle and release it upon completion of the script.

Example 23

local	hDBHandle	# Database handle
	*	variable
local	*	# Key to compare to
	keyValue	database field
local	nRet	# Return value from
	*	function call

```
hDBHandle = CreateDBHandle()
if (hDBHandle == (0)) then
```

FACE BLANK (USPTO)

Pat. No. 5870464, *
< invalid database handle, error

processing >

else

key Value = "(214)-612-5100"
nRet = RunQuery(hDBHandle, "phoneNumber",

" = ", key Value)

endif
nRet = DestroyDBHandle(hDBHandle)

The database handle is valid from the point in the script where the CreatedBHandle is called until the DestroyDBHandle is called. The database handle must be released before exiting the IIR script to ensure proper resource management. Check the database handle returned from CreatedBHandle for a valid, non-zero value. A return value of 0 indicates that the system was unable to allocate access to the customer database and should be handled as an error condition.

The RunQuery function locates a record or records in the database. This function has four parameters: database handle, database field name, query operator (one of "< ", "< = ", "= ", "> ", "> = "), and key value to compare to database values in the specified field name. The following Example 24 illustrates the RunQuery function:

Example 24

```
RunQuery(hDBHandle, "accountNo.", " > = ", "123456789")
```

The literal string "accountNo" must be one of the field names defined through the database administration tool. This string is the field name stored in the database itself. The operator in this case is the literal string "> =" (greater than or equal to), while the comparison key value is the literal string "123456789."

This line directs IIR to locate the first record in the database organized by accountNo (ascending order) where "accountNo" is greater than or equal to "123456789." If the function returns 0, the query was successful.

When a query has succeeded, the database handle represents the location of the record found. Various Set . . . and Get . . . field functions can then be used to update or read the customer's record. The RunQuery operation provides no combination logic, which means that no way exists of specifying that (field1 = KEY1 And field2 < = KEY2). If you require this type of logic, position the

THIS PAGE BLANK (USPTO)

database handle on a record using the field comparison that will result in the fewest number of matches. For example, in the example above field1 = KEY1 would provide the most restrictive criteria.

Once the database pointer has been positioned on this record, all records that have field1 = KEY1 can be cycled through using the MoveNextRecord function, while comparing field2 for being < = Key2 manually in the script. That logic would look similar to the Example 25 below:

Example 25

```
nRet = RunQuery( hDBHandle, "field1", " = ", "12345"
if (nRet == 0) then

    Do
    if (GetStringFieldValue( hDBHandle, "field2")

        < = "6789") then

        < process the customer record >
        Update Record( hDBHandle)

    endif
    nRet = MoveNextRecord( hDBHandle)
    fld1Tmp = GetStringFieldValue( hDBHandle,

        "field1")

    Loop Until ((nRet < > 0) And (fld1Tmp == "12345"))

endif
```

Always check the result of the RunQuery function call. If the result is non-zero, the query was unsuccessful and the current record is not defined. As mentioned above, you must test the second field separately. Use the function GetStringFieldValue in combination with the field2 to check for the second condition (field2 < = KEY2).

The function MoveNextRecord moves the database handle to the next record in ascending order of the field used in the RunQuery invocation, in this case field1. This next record must use field1 as well as field2 matching criteria. Although the query positions you at a matching record, it does not guarantee that the next or previous record match the same criteria, particularly in the case of the equal " = " comparison operator.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

MoveNextRecord and MovePreviousRecord operate according to the field files stored with the database. These index files organize the field in ascending order. In Example 26, a field1 index file might appear as:

le 26

.111 12121 12345 15555 21221 33221 . . .

In this example, the field index file has organized the values into ascending order for field1. The key field value "12345" appears in this list, with no duplicates of this entry. For this example, the MoveNextRecord would point to the record having field1 value "15555" as the current record, showing the necessity to check the primary key each move operation when the operator is "=".

As shown in the previous paragraph, the "=" (equal to) operator requires a retest of the primary criteria on each move. This requirement is not true, however, for the remaining operators ("<", "<=", ">", and ">=") due to the nature of their comparison. Because the index file is organized in ascending order, these operators guarantee that the next or previous element in sequential order will satisfy the criteria (assuming you have not reached the beginning or end of the index file).

When using the move functions (MoveNextRecord and MovePreviousRecord), you must consider how their use relates to the comparison operator used in the RunQuery statement. In the example index file from above which uses an operator of ">" (greater than) and a key value of "12345", the first element to satisfy this query would be "15555". Using the MoveNextRecord would place you at "21221", and so forth.

If you use the operator "<" (less than) and a key value of "12345", the operation will place the database handle position at key "12121", the first element less than the key value "12345". In this case the MovePreviousRecord would be the correct function to move to the next record that meets the "<" (less than) criteria. The following Table 15 summarizes the relationship between the operator and the correct move function to use. In addition, whether or not the user needs to retest for the primary criteria on each move is defined for each operator.

* Operators	Function for 'Next'	Retest Criteria Each Move?
" < " " > = "	Match	Not Required
" < " " < = "	MoveNextRecord	Not Required
" < " " > = "	MovePreviousRecord	Not Required
" = "	MoveNextRecord	Required

Appendix C, "Command Reference," contains a summary of all database functions. This summary includes function types Database and all functions that can operate on a database, as well as a short description of operation and syntax.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

Miscellaneous and System Functions

The classification of miscellaneous functions includes access to time and information from the system. All functions accessing the time information use a time handle.

A time handle need only be created once at the beginning of the IIR script using the CreateTimeHandle() function, and must be released before exiting the script using the DestroyTimeHandle() function. The following script segment in Example 27 shows the calls necessary to allocate the time handle and release it at the completion of the script.

```

Example 27      hTimeHandle
al              nRet
al              *
                # Time handle variable
                # Return value from
                function call

```

```

hTimeHandle = CreateTimeHandle (-)
if (hTimeHandle == 0) then
    hTimeHandle is invalid, error processing >
else
    nRet = GetCurrentTime (hTimeHandle)
    println "Current date and time is:" + GetAscTime(
        hTimeHandle 0
        [circle-solid]
        [circle-solid]
    endif
    nRet = DestroyTimeHandle (hDBHandle)

```

The time handle is valid from the point in the script where the CreateTimeHandle is called until the DestroyTimeHandle is called. The time handle must be released before exiting the IIR script to ensure proper resource management. It is important to check the time handle returned from CreateTimeHandle for a valid, non-zero value. A return value of 0 indicates that the system was unable to allocate a time handle. Once the time handle is obtained successfully, the user must use the GetCurrentTime function to actually update the instance time and date data. This call must be made each time the time and date information needs to be refreshed.

System functions are those inherent functions with the ICRL prefix. These functions currently include string manipulation functions such as ICRLStrLen.

A summary of all time functions can be found in Appendix D, including function types MiscTime and System. For detailed information on a specific miscellaneous or system function, see the detailed entry for that function in the command reference section.

Guidelines to Follow

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

is chapter has now defined all primary script components, including the
ting language constructs and all available functions. Please be certain
you have read and understood this information before you continue.

ne following section explains the components of a standard script. A
dard script includes access to information from the Route Request message,
arison of this data to that stored in the customer database, and finally a
rmination of where to route the call.

The following call center scenario will be helpful in understanding the
apt example.

A credit card issuer wants to take away the laborious task of manually
ifying new credit cards issued through the use of agents. The issuer
ermines that a match on the caller's ANI (Automatic Number Identification, or
ply put, the number from which they are calling) to the phone number in the
stomer database is sufficient for verification purposes, and can be automated
ing the IIR.

The IIR receives the adjunct route request, performs a lookup on the ANI
formation, and, if a match is found, updates the customer's record to reflect
rification on the card. If a match is not found, the IIR uses the RouteMore
apability to prompt the caller to enter other verification data (social
ecurity number, birth date, etc.).

Example 28 on the following page represents the IIR script that would handle
hese requirements. The example breaks down "blocks" of the script, A through G.
he text following the example defines important issues for each section and
describes the logic associated with that section.

```
Example 28
A.      global      $ 1-$ 0      $ 1 Is the call Identifier
          *           *           $ 0 the script name

          local      hTelHandle, hDBHandle      # declare
          *                                     vars for DB
          *                                     and tel
          *                                     handles

local queryRet, nRet, rteRet

local      ANIValue # ANI returned from tel

call

# declare vars to be used as

constants
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

local RETOK NOT FOUND, INVALID-HNDL, FAIL

RTE, QUERY-SSN

local VALID-VDN

3. # Initialize values for constants in the script
RETOK = 0 INVALID-HNDL = 0; FAIL-RTE = "3000"

QUERY-SSN = "3100"; VALID-VDN = "3200"

Create DataBase and Telephone handles
hDBHandle = CreatedBHandle()
hTelHandle = ICRLAtoi(\$ 1)

C. # Verify validity of database and telephony handles

before proceeding

If (hDBHandle == INVALID-HNDL Or hTelHandle ==

INVALID-HNDL) then

SetRouteSelected (hTelHandle, FAIL-RTE-; SetPriorityCall

(hTelHandle), (0)
rteRet = RouteFinal (hTelHandle)

else # 2valid handles, proceed with card verification

D. # the following line queries the DB for a match on

"PhoneNo" field to ANI

query Ret = RunQuery (hDBHandle, "PhoneNo", " = ",

GetCallingDevice (hTelHandle))

THIS PAGE BLANK (USPTO)

	Pat. No. 5870464, *
destVariable	Variable receiving the integer value returned from the record
hDBHandle	Variable that receives the allocated database handle
fieldName	Parameter of string type containing record field name to retrieve

Return Value

Returns the field's numeric value if successful, otherwise the function returns the integer - 99999.

Remarks

The GetNumericFieldValue function cannot be called before a database handle being obtained through the CreateDBHandle function, and a "current" record being defined through the RunQuery function call.

Example-GetNumericFieldValue

[variable declarations]

INVALID - HNDL = 0; RETOK = 0; END-OF-FILE = - 9; GOLD

= 1

```
hDBHandle = CreatedBHandle( )
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default
```

route processing

```
if ( hDBHandle == INVALID-HNDL then
```

```
    perform default route processing
```

```
else
```

```
    nRet = RunQuery( hDBHandle, "AccountNo", " = ",
```

```
        GetIVRDigits( hTelHandle))
```

```
    # while the field is equal to the key, set
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

acctFound field

do

SetStringFieldValue(hDBHandle,

"AcctNotes", "ACCT LOCATED")

UpdateRecord(hDBHandle)

nRet = MoveNextRecord(hDBHandle)

if (nRet == RETOK) then

If (GetNumericFieldValue(hDBHandle,

"CustClass") == GOLD) then

acctNo = GetStringFieldValue

(hDBHandle, "AccountNo")

endif

endif

while (nRet < > END-OF-FILE And acctNo ==

GetIVRDigits(hTelHandle))

endif

route call to default call processing VDN
DestroyDBHandle(hDBHandle)

Function GetNumIVRSets .

The GetNumIVRSets function returns the number of IVR "sets" available to this IIR script. This function, as well as the SetCurrentIVRSets function, is related to the RouteMore function explained in the Remarks section of this description.

Syntax

destVariable = GetNumIVRSets(hTelHandle)	
Part	Description

THIS PAGE BLANK (USPTO)

	Pat. No. 5870464, *
destVariable	Variable that receives the integer representing the number of IVR digit "sets".
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.

Return Value

This function returns an integer value that represents the number of IVR digit "sets". For an invalid telephony handle, the function returns the value - 4001.

Remarks

There are three functions related to the IVR set concept, RouteMore, GetNumIVRSets, and SetCurrentIVRSets. The RouteMore function allows an IIR script to perform multiple IVR digit queries with the caller without leading the control scope of the one IIR script.

The GetNumIVRSets allows the script to obtain the number of IVR digit sets that are potentially available, based upon the first script invocation as well as all RouteMore calls for more collected digits. The RouteMore call actually sends control back to a call prompting VDN to obtain more digits, where control is returned to the IIR script through the call vectoring adjunct command. Upon return from the RouteMore function, the most recent set of IVR collected digits is automatically available through the GetIVR functions (GetIVRType, GetIVRIndicator, and GetIVRDigits.)

The script can, however, also access any previous "set" of IVR data (within the scope of this one IIR script) by forcing the previous "set" of IVR data to focus with the SetCurrentIVRSets function.

Example-GetNumIVRSets
[variable declarations]
NO - IVRDATA = - 1; GET-PIN-VDN = "3200"; FIRST-SET = 0;

BOTH-SETS = 2

obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)
check to see if account number was entered by

caller, if so collect PIN number

THIS PAGE BLANK (USP10)

Pat. No. 5870464, *

```
if (GetIVRType(hTelHandle) < > NO-IVRDATA) then
```

```
    SetPriorityCall( hTelHandle, 0);
```

```
        SetRouteSelected( hTelHandle, GET-PIN-VDN)
```

```
RouteMore( hTelHandle )
```

```
if (GetNurnlVRSets( hTelHandle) == BOTH-SETS)
```

```
    then
```

```
    pinNum = GetIVRDigits( hTelHandle )
```

```
    SetCurrentIVRSets( hTelHandle, FIRST-SET)
```

```
    acctNum = GetIVRDigits( hTelHandle )
```

```
    Perform database match on acctNum/pinNum
```

```
        and process call
```

```
    endif
```

```
endif
```

Function GetSecond

The GetSecond function returns an integer representing the current second of the current minute based upon the most recent call to GetCurrentTime within the script.

Syntax

```
destVariable = GetSecond(hTimeHandle)
```

Part	Description
destVariable	Variable receiving an integer that is the current second (0-59.)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current second. For the case of an invalid time handle, the function returns - 5008.

THIS PAGE BLANK (USPTO)

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

```
Example-GetSecond
{variable declarations}
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle ()
# make sure the time handle is valid before

    accessing the data

if (hTimeHandle < > INVALID - HNDL) then

    # obtain the time data information
    GetCurrentTime( hTimeHandle )
    println "The current time is:" + GetAscTime

        (hTimeHandle)

    hr = GetHour( hTimeHandle); min =

        GetMinute(hTimeHandle);

    sec = GetSecond(hTimeHandle)
    println "From components hh:mm:ss " + hr + ":"

        + min + ":" + sec

    mo = GetMonth(hTimeHandle);

    day = GetDayOfMonth(hTimeHandle)

    yr = GetYear(hTimeHandle)
    println "From components, date is:" + mo + "/"

        + day + "/" + yr
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
println "Day of week:" + GetDayOfWeek
```

```
(hTimeHandle)
```

```
println "Daylight Savings Time (On/Off): " +
```

```
GetDst(hTimeHandle)
```

```
println "Day of year" +
```

```
GetDayOfYear(hTimeHandle)
```

```
DestroyTimeHandle( hTimeHandle)
```

```
endif
```

Function GetStringFieldValue

The GetStringFieldValue function returns the value for a specified string type field from the "current" database record. A current database record must be identified using the RunQuery operation before accessing fields within the record.

Syntax

```
destVariable = GetStringFieldValue(hDBHandle, fieldName)
```

Part	Description
destVariable	Variable receiving the string field result from the operation
hDBHandle	Variable that receives the allocated database handle
fieldName	Parameter of string type containing record field name to retrieve

Return Value

Returns the field's string value if successful, otherwise the function returns the string " < INVALID STRING > ".

Remarks

THIS PAGE BLANK (USPTO)

The GetStringFieldValue function cannot be called before a database handle being obtained through the CreateDBHandle function, and a "current" record being defined through the RunQuery function call.

Example-GetStringFieldValue

```
[variable declarations]
INVALID-HNDL = 0; RETOK = 0; END-OF-FILE = - 9
hDBHandle = CreateDBHandle()
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default

route processing

if ( hDBHandle == INVALID-HNDL) then

    perform default route processing

else

    nRet = RunQuery( hDBHandle, "AccountNo", " = ",

        GetStringFieldValue( hTelHandle))

    # while the field is equal to the key, set

        acctFound field

    do

        SetStringFieldValue( hDBHandle,

            "AcctNotes", "ACCT LOCATED")

        UpdateRecord( hDBHandle)
        nRet = MoveNextRecord( hDBHandle)
        if (nRet == RETOK) then

            acctNo = GetStringFieldValue

                (hDBHandle, "AccountNo")
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

endif

while (nRet < > END-OF-FILE And acctNo = =

GetIVRDigits(hTelHandle))

endif

route call to default call processing VDN
DestroyDBHandle(hDBHandle)

Function GetTrunk

The GetTrunk function returns the Trunk Group Number of the inbound originating trunk. This information is available only for calls that are originated over non-PRI lines. The GetCallingDevice function is used to retrieve information for calls originated over PRI lines or on-PBX extensions. This parameter is obtained from the Route Request Service message (Version 2), trunk field.

Syntax

destVariable = GetTrunk(hTelHandle)	
Part	Description
destVariable	Variable that receives the string returned from this function.
h TelHandle	Telephony handle obtained from 1CRLAtoi (\$ 1) function call.

Return Value

This function returns a string expression that represents the originating trunk group number. If the return value is a zero length string, no trunk information was received for this call. For an invalid telephony handle, the string "NULL-POINTER" is returned.

Remarks

The trunk field in the Route Request Service message is an optional component of this message, therefore it may not exist for each call. The trunk field is mutually exclusive with the callingDevice field accessed through the GetCallingDevice function. This means one or the other of callingDevice or trunk are available in the message, but never both.

Example-GetTrunk

[variable declarations]
obtain telephony handle by converting call ID to

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

numeric value

```
hTelHandle = ICRLAtoi ( $ 1 )
ANI = GetCallingDevice(hTelHandle)
# checking for a zero length string determines
```

whether the field is in the message

```
if (ANI < > "") then
```

```
    println "The calling device (ANI) is " + ANI
```

```
else
```

```
    # the trunk data will be available when the
```

```
        calling device isn't
```

```
    trunkID = GetTrunk( hTelHandle)
    println "The inbound trunk group ID = " +
```

```
        trunkID
```

```
endif
```

Function GetVDN

The GetVDN function returns the vector directory number (VDN) of the VDN that first handled the call on the switch. This parameter is obtained from the Route Request Service message (Version 2), currentRoute field.

Syntax

```
destVariable = GetVDN(h TelHandle)
```

Part	Description
destVariable	Variable that receives the string returned from this function.
h TelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

THIS PAGE BLANK (USPTO)

Return Value

If the telephony handle is valid, the function returns the currentRoute member of the

RouteRequestExt structure as a string, otherwise the string "NULL-POINTER" is returned.

Remarks

The currentRoute field in the Route Request Service message is a mandatory component of this message, meaning it will always be available upon receipt of a route request. This field is often used in the same manner as DNIS on an off-PBX inbound call.

Example-GetVDN

{variable declarations}
obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)
if the VDN is equal to accounting DNIS, route to

acct hunt group extension

if (GetVDN(hTelHandle) == "2222") then

SetRouteSelected(hTelHandle, "3000")

else # if VDN is rmarketing DNIS, route to mktg hunt

group extension
if (GetVDN(hTelHandle) == "3333") then

SetRouteSelected(hTelHandle, "3100")

endif

endif
define parameters required to route call
SetPriorityCall(hTelHandle, 0)
RouteFinal (hTelHandle)

THIS PAGE BLANK (USPTO)

Function GetYear

The GetYear function returns an integer representing the current year of the century (i.e. 1995 will return 95) based upon the most recent call to GetCurrentTime within the script.

Syntax

```
destVariable = GetYear(hTimeHandle)
```

Part	Description
destVariable	Variable receiving an integer that is the current year of the century (0-99.)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current year of the century. For the case of an invalid time handle, the function returns -5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetYear

```
[variable declarations]
```

```
RETOK = 0
```

```
INVALID-HNDL = 0
```

```
# Create Time Handle
```

```
hTimeHandle = CreateTimeHandle ()
```

```
# make sure the time handle is valid before
```

```
accessing the data
```

```
if (hTimeHandle < > INVALID - HNDL) then
```

```
# obtain the time data information
```

```
GetCurrentTime( hTimeHandle )
```

```
println "The current time is: " + GetAscTime
```

```
(hTimeHandle)
```

THIS PAGE BLANK (USPTO)

```
hr = GetHour( hTimeHandle ); min =  
  
    GetMinute(hTimeHandle);  
  
sec = GetSecond(hTimeHandle)  
println "From components hh:mm:ss " + hr + ":"  
  
    + min + ":" + sec  
  
mo = GetMonth(hTimeHandle);  
  
    day = GetDayOfMonth(hTimeHandle)  
  
yr=GetYear(hTimeHandle)  
println "From components, date is: " + mo + "/"  
  
    + day + "/" + yr  
  
println "Day of week:" + GetDayOfWeek  
  
    (hTimeHandle)  
  
println "Daylight Savings Time (On/Off) : " +  
  
    GetDst(hTimeHandle)  
  
println "Day of year" +  
  
    GetDayOfYear(hTimeHandle)  
  
DestroyTimeHandle( hTimeHandle)  
  
endif
```

Function ICRLAtoi

THIS PAGE BLANK (USPTO)

The ICRLAtoi function returns an integer version of the string passed in as the paramter.

Syntax

Part	Description
destVariable	Variable receiving the integer representation of the string.
string	String parameter to convert to numeric format. The string passed here must contain numeric characters only (0-9) or an error will be returned.

Return Value

If successful, this function returns a numeric representation of the string parameter.

Remarks

NONE

Example-ICRLAtoi

[variable declarations]
search and extract string components from the

comma delimited file

hTelHandle = ICRLAtoi (\$ 1)

Process the call using the numeric telephony handle converted above.

Function ICRLLeft

The ICRLLeft function returns the leftmost (first in a sequence) number of characters specified from the string parameter and returns this string.

Syntax

Part	Description
destVariable	Variable receiving the sub-string extracted from leftmost numChars.
stringParam	String parameter to extract leftmost numChars from.

THIS PAGE BLANK (00710)

numChars

Pat. No. 5870464, *
Integer parameter defining how
many characters to extract.

Return Value

If successful, this function returns a string that is the leftmost numChars specified from the stringParam. A null string (0 length string) is returned if the function is unable to perform the extraction due to inconsistent or invalid parameters.

Remarks

NONE

Example-ICRLLeft

[variable declarations]

search and extract string components from the

comma delimited file

tstStr = "'This', 'is', 'the', 'data', 'from',

'the', 'file', end'"

Count = 0
do

locate the starting ' single quote of a word,

and assign tstStr to start there

tstStr = ICRLStrStr(tstStr,"")

Extract the rightmost characters, less the

starting ' single quote

tstStr = ICRLRight(tstStr, ICRLStrLen(tstStr)

- 1)

Now extract the word up to the next single

THIS PAGE BLANK (USPTO)

quote mark

```
subStr = IRLLeft(tstStr, ICRLStrindex(tstStr,
    "",0))
```

Reassign the test string to the rightmost

characters, minus the substring

```
tstStr = ICRLRight(tstStr, ICRLStrLen(tstStr) -
    (ICRLStrLen (subStr) + 1))
```

```
print subStr + ""
Count = Count + 1
```

loop until (subStr == "end" Or Count 7 = 10)

Function ICRLMid

The ICRLMid function returns a sub-string from a string defined by a starting character index and a length of the sub-string.

Syntax

destVariable = ICRLMid(stringParam, startChar, numChars)	
Part	Description
destVariable	Variable receiving the sub-string extracted from right most numChars.
stringParam	String parameter to extract right most numChars from.
startChar	Index of starting character within stringParam (0 based index.)
numChars	Integer parameter defining how many characters to extract.

Return Value

THIS PAGE BLANK (US-77)

If successful, this function returns a string that is the middle numChars specified starting at index startChar from the stringParam. A null string (0 length string) is returned if the function is unable to perform the extraction due to inconsistent or invalid parameters.

Remarks

The startChar parameter is a zero based index. The first character in a string is the 0th character. with the second character being the 1st, and so on. This is important to keep in mind when performing string manipulations, to assure the intended sub-string is obtained in the ICRLMid function call.

Example-ICRLMid

[variable declarations]

search and extract string components from the

comma delimited file

```
tstStr = "This is the test string."
subStr = ICRLMid(tstStr, 5, 6)
println "Sub String:" + subStr
< the resultant output would be "'Sub String:is the" >
The fifth character in the string is a blank, but 0
```

based the fifth character is the "i" in is.

Function ICRLRight

The ICRLRight function returns the right most (last in a sequence) number of characters specified from the string parameter and returns this string.

Syntax

destVariable = ICRLLeft(stringParam, numChars)	
Part	Description
destVariable	Variable receiving the sub-string extracted from right most numChars.
stringParam	String parameter to extract right most numChars from.
numChars	Integer parameter defining how many characters to extract.

Return Value

THIS PAGE BLANK (USP 10)

If successful, this function returns a string that is the right most numChars specified from the stringParam. A null string (0 length string) is returned if the function is unable to perform the extraction due to inconsistent or invalid parameters.

Remarks

NONE

Example-ICRLRight

[variable declarations]

search and extract string components from the

comma delimited file

tstStr = "'This', 'is', 'the', 'data', 'from',

'the', 'file', 'end'".

Count = 0

do

do # locate the starting ' single quote of a word,

and assign tstStr to start there

tstStr = ICRLStrStr(tstStr"")

Extract the rightmost characters, less the

starting ' single quote

tstStr = ICRLRight(tstStr, ICRLStrLen

(tstStr) - 1)

Now extract the word up to the next single

quote mark

subStr = ICRLLeft(tstStr, ICRLStrIndex(

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
tstStr, "", 0))

Reassign the test string to the rightmost

characters, minus the substring

tstStr = ICRLRight(tstStr, ICRLStrLen(tstStr)

- (ICRLStrLen(subStr) + 1))

print subStr + " "
Count = Count + 1

loop until (subStr == "end" Or Count >= 10)

Function ICRLStrCopy

The ICRLStrCopy function returns a persistent copy of the string passed in.

Syntax

destVariable = ICRLStrCopy(stringParam)	
Part	Description
destVariable	Variable receiving the copy of the stringParam.
stringParam	String parameter to copy.

Return Value

If successful, this function returns a string that is a copy of the stringParam. A null string (0 length string) is returned if the function is unable to perform the extraction due to inconsistent or invalid parameters.

Remarks

NONE

Example-ICRLStrCopy

[variable declarations]

search and extract string components from the

comma delimited file

THIS PAGE BLANK (USPTO)

```
tstStr = "This is the test string."
strCopy = ICRLStrCopy(tstStr)
```

Function ICRLStrIndex

The ICRLStrIndex function searches a string for a specified sub-string starting at a given index in the string, and if found, returns the index of the first character in the sequence.

Syntax

```
destVariable = ICRLStrIndex(stringParam, findStr, startIdx)
```

Part	Description
destVariable	Variable receiving the integer index of the sub-string or failure code.
stringParam	String variable being searched for occurrence of findStr.
findStr	String variable to locate in the stringParam.
startIdx	Index in stringParam (0 based) to start searching for findStr.

Return Value

If successful, this function returns an index to the findStr within the stringParam. If the findStr is not located in the stringParam, starting at the designated startIdx, then the function returns a - 1. The function also returns a - 1 in the case where the string parameters and start index have inconsistent or illegal data.

Remarks

The index returned from this function as well as the startIdx parameter are both 0 based. That is to say the first character index of any string is 0, rather than 1. If searching for the string "cat" in the string "raining cats and dogs", and giving a startIdx of 0 (start at the beginning of the string), the resulting index would be 8, even though the word cat begins with the ninth (9th) character. Once again, the indexes are 0 based, and are therefore counting the first character of a string as the 0th character.

This index can be used to select a sub-string from a super-string using functions such as ICRLMid.

Example-ICRLStrIndex

```
[variable declarations]
# search and extract string components from the
```

comma delimited file

THIS PAGE CONTAINS NO INFO

Pat. No. 5870464, *

```
tstStr    tstStr = "'This', 'is', 'the', 'data', 'from',
               'the', 'file', 'end'"

Count = 0
do
do        # locate the starting ' single quote of a word,
           and assign tstStr to start there

tstStr = ICRLStrStr( tstStr, '"')
# Extract the rightmost characters, less the
           starting ' single quote

tstStr = ICRLRight (tstStr, ICRLStrLen
                   ( tstStr ) - 1)

# Now extract the word up to the next single
           quote mark

subStr = ICRLLeft( tstStr, ICRLStrindex
                  ( tstStr, "'", 0))

# Reassign the test string to the rightmost
           characters, minus the substring

tstStr = ICRLRight( tstStr, ICRLStrLen( tstStr)
                  - (ICRLStrLen( subStr ) + 1))
print subStr + " "
Count = Count + 1
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
loop until (subStr == "end" Or Count >= 10)

Function ICRLStrLen

The ICRLStrLen function returns an integer representing the length of the string passed in as a parameter to the function.

Syntax

destVariable = ICRLStrLen(stringParam)	
Part	Description
destVariable	Variable receiving an integer that is the length of the string parameter.
stringParam	String variable for which the length is requested.

Return Value

If successful, this function returns an integer >= 0 that is the string length. If the stringParam is invalid, this function returns - 1.

Remarks

This function returns the length of the string, not counting the null terminator. For instance, the string "this" would have a string length of 4.

Example-ICRLStrLen

[variable declarations]

the following string length operation would return

a string length of (11)

```
strLen = ICRLStrLen( "This String")
```

Function ICRLStrStr

The ICRLStrStr function searches a string for a specified sub-string, and if found, returns a pointer into the search string where the sub-string was found.

Syntax

destVariable = ICRLStrStr(StringParam, findStr)	
Part	Description
destVariable	Variable receiving sub-string of

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

stringParam	String variable being searched for occurrence of findStr.
findStr	String variable to locate in the stringParam.

Return Value

If successful, this function returns a string that is the sub-string starting with the findStr parameter. If the findStr is not located in the stringParam then the function returns a 0 length string.

Remarks

NONE

Example-ICRLStrStr

[variable declarations]

Find the substring starting with the word

"Petrified"

resultStr = ICRLStrStr("Walk through the Petrified

Forest", "Petrified")

println "SubString is: " + resultStr
< this would result in the string "SubString is:

Petrified Forest" being displayed >

Function InsertRecord

The InsertRecord function adds a newly defined record to the database. This record's fields are defined using the SetStringFieldValue and SetNumericFieldValue functions, before insertion. The insertion of a record is dependent upon those fields defined as unique keys (through the database administrator) having unique values assigned to them before insertion.

Syntax

destVariable = InsertRecord(hDBHandle)	
Part	Description
destVariable	Variable receiving integer value

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
representing success/failure of
call
Variable that receives the
allocated database handle

hDBHandle

Return Value

Returns 0 if successful, for an invalid database handle the function returns - 5008. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve are covered in the IIR trouble shooting guide.

Remarks

The InsertRecord function cannot be called before a database handle being obtained through the CreateDBHandle function. The fields defined as "primary" keys through the IIR database administrator, need to have been set to unique values using the SetStringFieldValue, as well as all other fields being defined before the insertion occurs.

Example-InsertRecord
[variable declarations]
INVALID-HNDL = 0; RETOK = 0
hDBHandle = CreateDBHandle()
hTelHandle = ICRLAtOI (\$ 1)
If unable to create a database handle, do default

route processing

if (hDBHandle == INVALID - HNDL) then

perform default route processing

else

nRet = RunQuery(hDBHandle, "AccountNo", " = ",

GetIVRDigits(hTelHandle))

If the Query returned return OK, then modify

key fields and save as new record

if (nRet == RETOK) then

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
SetStringFieldValue (hDBHandle,

    "AccountNo", "33333333")

SetShingFieldValue (hDBHandle, "LastName",

    "Smith")

SetStringFieldValue (hDBHandle,

    "FirstName", "James")

If (InsertRecord( hDBHandle ) < > RETOK )

    then
        process error on inserting new record

endif

endif
```

```
endif
DestroyDBHandle ( hDBHandle )
```

Function MoveNextRecord

The MoveNextRecord function reassigned the hDBHandle to the next record in the database, based upon the field index most recently referenced in a RunQuery command. For example, if a record is located using the RunQuery function using "AccountNo" as the comparison field name, the MoveNextRecord moves to the next record in the AccountNo field index file, or returns an error for end of file.

Syntax

destVariable = MoveNextRecord(hDBHandle)	
Part	Description
destVariable	Variable receiving integer value representing success/failure of call.
hDBHandle	Variable that receives the allocated database handle

THIS PAGE BLANK (USPTO)

Return Value

Returns 0 is successful, for an invalid database handle the function returns - 5008. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve are covered in the IIR trouble shooting guide. When there is no next record (end of file), this function will return Btrieve error - 9.

Remarks

The MoveNextRecord function cannot be called before a database handle being obtained through the CreatedBHandle function, and a "current" record being defined through the RunQuery function call.

Example-MoveNextRecord

```
[variable declarations]
INVALID -HNDL = 0; RETOK = 0; END-OF-FILE = - 9
hDBHandle = CreatedBHandle
( hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default
```

```
route processing
```

```
if ( hDBHandle = = INVALID-HNDL) then
```

```
perform default mute processing
```

```
else
```

```
nRet = RunQuery( hDBHandle, "AccountNo", " = ",
```

```
GetIVRDigits( hTelHandle))
```

```
# while the field is equal to the key, set
```

```
acctFound field
```

```
do
```

```
SetStringFieldValue( hDBHandle,
"AcctNotes", "ACCT LOCATED")
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
UpdateRecord( hDBHandle )
nRet = MoveNextRecord(hDBHandle)
if (nRet == RETOK ) then
```

```
    acctNo = GetStringFieldValue
```

```
        (hDBHandle, "AccountNo")
```

```
endif
```

```
while (nRet < > END-OF-FILE And acctNo ==
GetIVRDigits( hTelHandle))
```

```
endif
oute call to default call processing VDN
estroyDBHandle( hDBHandle)
```

Function MovePreviousRecord

The MovePreviousRecord function reassigns the hDBHandle to the preceding record in the database, based upon the field index most recently referenced in a RunQuery command. For instance, if a record has been located using the RunQuery function using "AccountNo" as the comparison field name, the MovePreviousRecord will move to the preceding record in the AccountNo field index file, or return an error for end of file.

Syntax

destVariable = MovePreviousRecord(hDBHandle)	
Part	Description
destVariable	Variable receiving integer value representing success/failure of call.
hDBHandle	Variable that receives the allocated database handle

Return Value

Returns 0 if successful, for an invalid database handle the function returns - 5008. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve are covered in the IIR trouble shooting guide. When there is not preceding record (beg of file), this function will return Btrieve error - 9.

THIS PAGE BLANK (USPTO)

Remarks

The MovePreviousrecord function cannot be called before a database handle being obtained through the CreateDBHandle function, and a "current" record being defined through the RunQuery function call.

Example-MovePreviousRecord

```
{variable declarations}
INVALID HNDL = 0; RETOK = 0; BEG-OF-FILE = - 9
hDBHandle = CreateDBHandle()
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default

    route processing

if ( hDBHandle == INVALID - HNDL then

    perform default route processing

else

    nRet = RunQuery( hDBHandle, "AccountNo", " < ",

        GetIVRDigits( hTelHandle)

    # while the field is less than the entered

        digits

    do

        SetStringFieldValue( hDBHandle,

            "AcctNotes", "ACCT LOCATED")

        UpdateRecord( hDBHandle)
        nRet = MoveProvioueRecord( hDBHandle)

    while (nRet < > BEG-OF-FILE)

endif
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

oute call to default call processing VDN
estroyDBHandle(hDBHandle)

Function QueryAgentState

The QueryAgentState function obtains information on the state of an agent on the DEFINITY G3 switch. The agents device ID (extension for splits/login ID for kills based routing) and agent split/skill are used to obtain the information. After calling this function, various Get . . . functions related to the agent status can be called (GetAgentAvailable, GetAgentState, GetAgentWorkMode, and GetAgentTalkState.)

Syntax

```
destVariable = QueryAgentState(hTelHandle, agDevice, agSplit)
```

Part	Description
destVariable	Variable that receives the success/failure notification for this call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
agDevice	Mandatory parameter defining a valid agent extension.
agSplit	Mandatory parameter defining a valid ACD split/skill.

Return Value

This function returns an integer value that represents the success/failure of the call. If the call succeeds, a 0 is returned, for an invalid telephony handle, - 4001 is returned. If there are inconsistencies between parameters or an invalid parameter is passed, - 4002 is returned. For failures at the TSERVER, see the trouble shooting section for possible error Return Values.

Remarks

The agDevice parameter can be the agent login-ID when using skills based routing on the switch. Likewise, the agSplit can be one of the skill extensions assigned to the agent (i.e. the hunt group extensions of the related skills.)

Example-QueryAgentState

```
[variable declarations]
SKILL-HUNT = "2800"
hTelHandle = ICRLAtoi ($ 1)
# "3600" represents a specific agents login-ID
nRet = QueryAgentState( hTelHandle, "3600",
```

SKILL-HUNT)

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
(GetAgentAvailable( hTelHandle ) ) then
```

```
    send caller to this specific agent
```

```
lse
```

```
    send caller to general queue for handling
```

```
endif
```

Function RouteFinal

The RouteFinal function is one of three route specification functions (RouteMore, RouteFinal, and RouteUnknown.) An IIR script uses this function to route a call for the last time in the context of the current script execution. Any use of the telephony handle after execution of this function will return an error.

Syntax

destVariable = RouteFinal(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing success/failure of call. Any non-zero failure code received can include any of the error codes defined for the telephony functions in the trouble-shooting guide.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.

Return Value

If successful the function returns 0, for an invalid telephony handle - 4001. All other non-zero error codes can be referenced to the telephony function error codes in the trouble-shooting guide.

Remarks

Performing the RouteFinal operation implies that the FINAL destination for the call has been determined for the execution of this script. The RouteFinal command inherently releases the telephony handle obtained from the call to ICRLAtoi(\$ 1). Any reference to the telephony handle after the RouteFinal call in the script will result in an invalid telephony handle error being returned.

THIS PAGE BLANK (USPTO)

Syntax

Part	Description
destVariable	Variable receiving an integer that is the current day of the week (1 -7) where the first day of the week is Sunday (1 = Sunday, 2 = Monday, etc.)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle a call.

Return Value

If successful, this function returns an integer defining the current day of the week; if invalid time handle, the function returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetDayOfWeek

```
[variable declarations]
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
```

```
hTimeHandle = CreateTimeHandle()
# make sure the time handle is valid before
```

```
accessing the data
```

```
if (hTimeHandle < > INVALID-HNDL) then
```

```
# obtain the time data information
GetCurrentTime( hTimeHandle )
println "The current time is:" + GetAscTime
```

```
(hTimeHandle)
```

```
hr = GetHour( hTimeHandle ); min =
```

THIS PAGE INTENTIONALLY LEFT BLANK (UCPT0)

Pat. No. 5870464, *
GetMinute(hTimeHandle);

```
sec = GetSecond(hTimeHandle)
println "From components hh:mm:ss:" + hr + ":" + min

+ ":" + sec
```

```
mo = GetMonth(hTimeHandle);
```

```
day = GetDayOfMonth(hTimeHandle)
```

```
yr=GetYear(hTimeHandle)
println "From components, date is:" + mo + ":" + day

+ "/" + yr
```

```
println "Day of week:" + GetDayOfWeek( hTimeHandle)
println "Daylight Savings Time (On/Off):" +
```

```
GetDst(hTimeHandle)
```

```
println "Day of year:" + GetDayOfYear(hTimeHandle)
DestroyTimeHandle( hTimeHandle )
```

```
endif
```

Function GetDayOfYear

The GetDayOfYear function returns an integer representing the current day of the year (1-366); the base or first day January 1 (1 = Jan 1) based upon the most recent call to GetCurrentTime within the script.

Syntax

destVariable = GetDayOfYear(h TimeHandle)	
Part	Description
destVanable	Variable receiving an integer that is the current day of the year (1 -366) where the first day of the year is Jan 1 (1 = Jan 1.)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

THIS PAGE BLANK (USPTO)

Return Value

If successful, this function returns an integer defining the current day of the year. For the case of an invalid time handle, the function returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetDayOfYear

```
{variable declarations}
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle()
# make sure the time handle is valid before

    accessing the data

if (hTimeHandle < > INVALID-HNDL) then

    # obtain the time data information
    GetCurrentTime( hTimeHandle )
    println "The current time is)" + GetAscTime(

        hTimeHandle )

    hr = GetHour( hTimeHandle ); min =

        GetMinute(hTimeHandle);

    sec = GetSecond(hTimeHandle)
    println "From components hh:mm:ss : " + hr +

        ":" + min + ":" + sec

    mo = GetMonth(hTimeHandle);

    day = GetDayOfMonth(hTimeHandle)
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
yr = GetYear(hTimeHandle)
println "From components, date is:" mo + "/" +
```

```
day + "/" + yr
```

```
println "Day of week:" + GetDayOfWeek
```

```
(hTimeHandle)
```

```
println "Daylight Savings Time (On/Off): " +
```

```
GetDst(hTimeHandle)
```

```
println "Day of year:" +
```

```
GetDayOfYear(hTimeHandle)
```

```
DestroyTimeHandle( hTimeHandle )
```

```
endif
```

Function GetDst

The GetDst function returns an integer representing the current status of daylight savings time (on or off). This information is based upon the most recent call to GetCurrentTime within the script.

Syntax

```
destVariable = GetDst(hTimeHandle)
```

Part	Description
destVariable	Variable receiving an integer that is the state of daylight savings time (0 = Off, 1 = On).
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current daylight savings time state. For the case of an invalid time handle, the function

THIS PAGE BLANK (USPTO)

returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetDst

```
{variable declarations}
```

```
RETOK = 0
```

```
INVALID-HNDL = 0
```

```
# Create Time Handle
```

```
hTimeHandle = CreateTimeHandle()
```

```
# make sure the time handle is valid before
```

```
accessing the data
```

```
if (hTimeHandle < > INVALID-HNDL) then
```

```
# obtain the time data information
```

```
GetCurrentTime( hTimeHandle )
```

```
println "The current time is : " + GetAscTime
```

```
(hTimeHandle )
```

```
hr = GetHour( hTimeHandle ); min =
```

```
GetMinute(hTimeHandle);
```

```
sec = GetSecond(hTimeHandle)
```

```
println "From components hh:mm:ss: " + hr + ":"
```

```
+ min + ":" + sec
```

```
mo = GetMonth(hTimeHandle);
```

```
day = GetDayOfMonth(hTimeHandle)
```

```
yr = GetYear(hTimeHandle)
```

```
println "From components, date is : " + mo +
```

THIS PAGE UNCLASSIFIED (USPTO)

Pat. No. 5870464, *
"/" + day + "/" + yr

println "Day of week:" + GetDayOfWeek

(hTimeHandle)

println "Daylight Savings Time (On/Off): " +

GetDst(hTimeHandle)

println "Day of year:" +

GetDayOfYear(hTimeHandle)

DestroyTimeHandle(hTimeHandle)

endif

Function GetHour

The GetHour function returns an integer representing the current hour based upon the most recent call to GetCurrentTime within the script.

Syntax

destVariable = GetHour(hTimeHandle)	Description
Part	Variable receiving an integer
destVariable	that is the current hour (0-23)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current hour. For the case of an invalid time handle, the function returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and data information through a call to GetCurrentTime before calling this function. The information returned by this call will be based

THIS PAGE DELETED (USPTO)

Pat. No. 5870464, *

n the last call to GetCurrentTime in the script.

Example-GetHour

[variable declarations]

RETOK = 0

INVALID-HNDL = 0

Create Time Handle

hTimeHandle = CreatTimeHandle()

make sure the time handle is valid before

accessing the data

if (hTimeHandle < > INVALID-HNDL) then

obtain the time data information
GetCurrentTime(hTimeHandle)

println "The current time is : " + GetAscTime(

hTimeHandle)
hr = GetHour(hTimeHandle); min

GetMinute(hTimeHandle);

sec = GetSecond(hTimeHandle)
println "From components hh:mm:ss : " + hr +

":" + min + ":" + sec

mo = GetMonth(hTimeHandle);
day = GetDayOfMonth(hTimeHandle)
yr = GetYear(hTimeHandle)
println "From components, date is : " + mo +

"/" + day + "/" + yr

println "Day of week : " + GetDayOfWeek

(hTimeHandle)

println "Daylight Savings Time (On/Off) : " +

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
GetDst(hTimeHandle)

printin "Day of year" +

GetDayOfYear(hTimeHandle)

DestroyTimeHandle(hTimeHandle)

endif

Function GetIncoming UIIData

The GetIncomingUIIData function returns the user-to-user information data.
The type of user-to-user information, obtained through the GetIncomingUIType,
indicates the presence of user-to-user information (UII) including the UII Data.

Syntax

destVariable = GetIncomingUIIData(hTelHandle)	Description
Part	Variable that receives the string
destVariable	representing the UII Data.
hTelHandle	Telephony handle obtained from ICRLAtol(\$ 1) function call.

Return Value

This function returns a string value that represents the user-to-user
information data. For an invalid telephony handle, the function returns the
string "NULL-POINTER".

Remarks

The userinfo.type, accessed through the function GetIncomingUIType,
determines whether or not any user-to-user information actually exist. If the
userinfo type is none (- 1), there is no user-to-user information for this
call.

All userinfo parameters contained in the Route Request message are ATT
private data elements. The specific field in the Route Request message (Version
2) is private data ATTUserToUserInfo.data.value.

Example-GetIncomingUIIData
[variable declarations]

NO-UII = -1; UII-ASCII = 4

obtain telephony handle by converting call ID to

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

numeric value

```
hTellHandle = ICRLAtol( $ 1 )  
# check to see if the caller entered the expected
```

DTMF digits

```
if (GetIncomingUUType( hTellHandle == NO-UUI ) then
```

```
.  
Process Without user-to-user information  
.
```

else

```
uuiLen = GetIncomingUULength( hTelHandle )  
uuiData = GetIncomingUUData( hTelHandle )  
if ( uuiLen == 9 ) then
```

Process the uuiData as a SSN from adjacent

switch

endif

endif

Function GetIncomingUULength

The GetIncomingUULength function returns the user-to-user information length. The type of user-to-user information, obtained through the GetIncomingUUType, indicates the presence of user-to-user information (UUI) including the UUI Length.

Syntax

destVariable = GetIncomingUULength(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing the UUI Length.

THIS PAGE BLANK (USPTO)

hTelHandle

Pat. No. 5870464, *
Telephony handle obtained from
ICRLAtoi(\$ 1) function call.

Return Value

This function returns an integer value that represents the user-to-user information length. For an invalid telephony handle, the error - 4001 (ERR-INVALID-TEL-HANDLE) is returned.

Remarks

The userInfo type, accessed through the function GetIncomingUUType, determines whether or not any user-to-user information actually exists. If the userInfo type is none (- 1), no user-to-user information for this call.

All userInfo parameters contained in the Route Request message are ATTivate data elements. The specific field in the Route Request message (Version) is private data ATTUserToUserInfo.data.length.

Example-GetIncomingUULength
[variable declarations]

NO-UUI = -1; UUI-ASCII = 4

obtain telephony handle by converting call ID to

numeric value

hTelHandle ICRLAtoi(\$ 1)
check to see if the caller entered the expected

DTMF digits

if (GetIncomingUUType(hTelHandle) == NO-UUI)

then
Process without user-to-user information

else

uuiLen = GetIncomingUULength(hTelHandle)
uuiData = GetIncomingUUData(hTelHandle)
if (uuiLen == 9) then

Process the uuiData as a SSN from adjacent

THIS PAGE INTENTIONALLY LEFT BLANK (USPTO)

Pat. No. 5870464, *
switch

endif

endif

Function GetIncomingUIType

The GetIncomingUIType function returns the user-to-user information type. The type of user-to-user information indicates the presence of user-to-user information (UII) and the format of the data portion of the message.

Syntax

destVariable = GetIncomingUIType(hTelHandle)	Description
Part	Variable that receives the
destVariable	integer representing the UII
	Type.

1	UII-NONE (No UII data specified)
0	UII-USER SPECIFIC
1	UII-IA5-ASCII

hTelHandle	Telephony handle obtained from ICRLAtol(\$ 1) function call.
------------	---

Return Value

This function returns an integer value that represents the user-to-user information type. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

The userInfo.type determines whether or not any user-to-user information actually exist. If the userInfo type is none (- 1), there is no user-to-user information for this call.

All userInfo parameters contained in the Route Request message are ATT private data elements.

The specific field in the Route Request message (Version 2) is private data ATTUserToUserInfo.type.

THIS PAGE BLANK (USPTO)


```
Example-GetIncomingUIType
[variable declarations]
NO-UII = -1; UII-ASCII = 4
# obtain telephony handle by converting call ID to
```

```
numeric value
```

```
hTelHandle = ICRLAtOI( $ 1 )
# check to see if the caller entered the expected
```

```
DTMF digits
```

```
if (GetIncomingUIType( hTelHandle ) NO-UII then
```

```
Process without user-to-user information
```

```
else
```

```
uiiLen = GetIncomingUIILength( hTelHandle )
uiiData = GetIncomingUIIData( hTelHandle )
if ( uiiLen == 9 ) then
```

```
Process the uiiData as a SSN from adjacent
```

```
switch
```

```
endif
```

```
endif
```

Function GetIVRCollectVDN

The GetIVRCollectVDN function returns the VDN that collected the code/digits entered by the caller through the G3 call prompting feature of the collected digits feature.

Syntax

```
destVariable = GetIVRCollectVDN(hTelHandle)
Part          Description
destVariable  Variable that receives the string
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
representing the VDN which
collected the code/digits in the
userEnteredCode data.
Telephony handle obtained from
ICRLAtoi(\$ 1) function call.

hTelHandle

return Value

This function returns a string value that represents the VDN having collected user entered code/digits. For an invalid telephony handle, the function returns the string "NULL-POINTER".

Remarks

The userEnteredCode type determines whether or not any code/digits actually exist and whether a CollectVDN exists. If the userEnteredCode type is none (- , no user-entered code/digits exist for this call. All userEnteredCode parameters in the Route Request message are ATT private data elements. The specific field in the Route Request message (Version 2) is private data. *TUserEnteredCode.collectVDN.

Example-GetIVRCollectVDN
{variable declarations}

NO-IVRDIGITS = -1; COLLECTED = 0; ENTERED = 1
obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi(\$ 1)
check to see if the caller entered the expected

DTMF digits

if (GetIVRType(hTelHandle) == NO-IVRDIGITS) then

Route to operator

else

if (GetIVRIndicator(hTelHandle) == COLLECTED)

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
then
  IVRDigits = GetIVRDigits( hTelHandle )
  collectVDN = GetIVRCollectVDN(hTelHandle)
  if (collectVDN == "3200") then
```

Process IVR Digits as Social Security

Number

else

if (collectVDN == "3300") then

Process IVR Digits as Account

Number

endif

endit

endit

endif

Function GetIVRDigits

The GetIVRDigits function returns the user collected code/digits. This data is that entered by the user through collect digit commands in vector processing on the G3 switch.

Syntax

destVariable = GetIVRDigits(hTelHandle)	
Part	Description
destVariable	Variable that receives the string representing the user entered code/digits.
hTeHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

return Value
This function returns a string value that represents the user collected
/digits. For an invalid telephony handle, the function returns the string
"L-POINTER".

Remarks
The userEnteredCode type determines whether or not any code/digits actually
exist. If the userEnteredCode type is none (- 1), there is no user entered
code/digits for this call.

All userEnteredCode parameters contained in the Route Request message are ATT
ivate data elements. The specific field in the Route Request message (Version
is private data ATTUserEnteredCode.data.

Example-GetIVRDigits
[variable declarations]
NO-IVRDIGITS = -1; COLLECTED = 0; ENTERED = 1
obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi(\$ 1)
check to see if the caller entered the expected

DTMF digits

if (GetIVRType(hTelHandle) == NO-IVRDIGITS) then

Route to operator

else

if (GetIVRIndicator(hTelHandle) == COLLECTED)

then
IVRDigits = GetIVRDigits(hTelHandle)
Process the caller entered IVR digits as

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
collected
.
```

```
endif
```

```
endif
```

Function GetIVRIndicator

The GetIVRIndicator function returns the user collected digits indicator. This indicator defines whether the data was collected or entered.

Syntax

```
destVariable = GetIVRIndicator(hTelHandle)
Part
destVariable      Description
                   Variable that receives the
                   integer representing the
                   indicator for the user-entered
                   digits.
```

```
0 Collect
1 Entered
```

hTelHandle Telephony handle obtained from
ICRLAtoi(\$ 1) function call.

Return Value

This function returns an integer value that represents the user collected code indicator. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

All userEnteredCode parameters contained in the Route Request message are ATT private data elements. The specific field in the Route Request message (Version 2) is private data ATTUserEnteredCode.indicator.

Example-GetIVRIndicator

```
{variable declarations}
NO-IVRDIGITS = -1; COLLECTED = 0; ENTERED = 1
# obtain telephony handle by converting call ID to
```

THIS PAGE BLANK (USPTO)

numeric value

```
hTelHandle = ICRLAtoi( $ 1 )  
# check to see if the caller entered the expected
```

DTMF digits

```
if (GetIVRType(hTelHandle) == NO-IVRDIGITS) then
```

```
    .  
    Route to operator  
    .
```

```
else
```

```
    if (GetIVRIndicator(hTelHandle) == COLLECTED)
```

```
        then  
        IVRDigits = GetIVRDigits( hTelHandle )  
        Process the caller entered IVR digits as
```

```
            collected  
            .  
            .
```

```
    endif
```

```
endif
```

Function GetIVRType

The GetIVRType function returns the user collected digits type. This type defines the method used to define the user collected digits, otherwise known as IVR Digits.

Syntax

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
destVariable = GetIVRType(hTelHandle)
Part          Description
destVariable  Variable that receives the
              integer representing user
              collected digit type

1 None (no code/digits
were
```

entered by the user)

```
0 Any
1 Login Digits
2 Call Prompter
3 Database Provided
4 Tone Detector
```

```
hTelHandle    Telephony handle obtained from
              ICRLAtoi($ 1) function call.
```

Return Value

This function returns an integer value that represents the user collected code type. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

If the user collected code type is None (- 1), then there is no information in the userEnteredCode for this call.

All userEnteredCode parameters contained in the Route Request message are ATT private data elements. The specific field in the Route Request message (Version 2) is private data ATTUserEnteredCode.type.

```
Example-GetIVRType
[variable declarations]
NO-IVRDIGITS = -1
# obtain telephony handle by converting call ID to
```

numeric value

```
hTelHandle = ICRLAtoi( $ 1 )
# check to see if the caller entered the expected
```

DTMF digits

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

if (GetIVRType(hTelHandle) == NO-IVRDIGITS) then

Route to operator

else

IVRDigits = GetIVRDigits(hTelHandle)
Process the caller entered IVR digits

endif

Function GetLookAheadHours

The GetLookAheadHours function returns the lookahead interflow Hour. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another conditionally (that is to say the destination switch can either accept or decline the call.) The lookahead hour is the hour part of the current time when the call was interflowed from the first switch. The lookahead interflow information is filled in by the switch that interflows the call.

Syntax

destVariable = GetLookAheadHours(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing the Interflow Hour.
hTelHandle	Telephony handle obtained from ICRLAtoi(\$) function call.

Return Value

This function returns an integer value that represents the lookahead interflow hour in military form. The lookahead interflow hour will be present if the lookahead interflow type is a valid one. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

THIS PAGE BLANK (USPTO)

Remarks

The lookahead interflow hour is one of six lookahead interflow information components. The remaining five are lookahead type, lookahead source VDN, lookahead priority, lookahead minutes, and seconds. The lookahead interflow type can be used to determine whether any of the interflow information exists for a given call.

All lookahead interflow parameters contained in the Route Request message are ATT private data elements.

Example-GetLookAheadHours

[variable declarations]

NO-IFLOW = -1

obtain telephony handle by converting call ID to

numeric value

```
hTelHandle = ICRLAtoi( $ 1 )
lkahdType = GetLookAheadType( hTelHandle )
lkahdHours = GetLookAheadHours( hTelHandle )
lkahdMinutes = GetLookAheadMinutes( hTelHandle )
lkahdSeconds = GetLookAheadSeconds( hTelHandle )
# If lookahead type is not no interflow, output the
```

time of the switch interflow

```
if (lkahdType < > NO-IFLOW) then
```

```
println "Call interflowed at: " + lkahdHours +
```

```
":" + lkahdMinutes + ":" lkahdSeconds
```

```
endif
```

Function GetLookAheadMinutes

The GetLookAheadMinutes function returns the lookahead interflow Minute. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another conditionally (that is to say the destination switch can either accept or decline the call.) The lookahead minute is the minute part of the current time when the call was interflowed from the first switch. The lookahead interflow information is filled in by the switch that interflows the call.

THIS PAGE BLANK (USPTO)

Syntax

destVariable = GetLookAheadMinutes(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing the Interflow Minute.
hTelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

Return Value

This function returns an integer value that represents the lookahead interflow minute. The lookahead interflow hour will be present if the lookahead interflow type is a valid one. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

The lookahead interflow minute is one of six lookahead interflow information components. The remaining five are lookahead type, lookahead priority, lookahead source VDN, lookahead hours, and seconds. The lookahead interflow type can be used to determine whether any of the interflow information exists for a given call.

All lookahead interflow parameters contained in the Route Request message are ATT private data elements.

Example-GetLookAheadMinutes

```
[variable declarations]
```

```
NO-IFLOW = -1
```

```
# obtain telephony handle by converting call ID to
```

```
numeric value
```

```
hTelHandle = ICRLAtoi( $ 1 )
lkahdType = GetLookAheadType( hTelHandle )
lkahdHours = GetLookAheadHours( hTelHandle )
lkahdMinutes = GetLookAheadMinutes( hTelHandle )
lkahdSeconds = GetLookAheadSeconds( hTelHandle )
# If lookahead type is not no interflow, output the
```

```
time of the switch interflow
```

```
If (lkahdType < > NO-IFLOW) then
```

```
println "Call interflowed at : " + lkahdHours +
```

THIS PAGE BLANK (USPTO)

```
":" + lkahdMinutes + ":" + lkahdSeconds
```

```
endif
```

Function GetLookAheadPriority

The GetLookAheadPriority function returns the Interflow Priority. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another conditionally (i.e., the destination switch can either accept or decline the call.) The priority is that assigned to the call while in queue on the first switch if interflowed while in queue. The information is filled in by the switch that interflows the call.

Syntax

destVariable = GetLookAheadPriority(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing the Interflow priority which can be one of the following: 0 Call was not in queue 1 Low 2 Medium 3 High 4 Top
hTelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

Return Value

This function returns an integer value that represents the lookahead interflow priority. The lookahead interflow priority is valid if the interflow type is one of the valid types. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

The lookahead interflow priority is one of six lookahead interflow information components. The remaining five are lookahead type, lookahead source VDN, lookahead hours, minutes, and second. The lookahead interflow type accessed via the GetLookaheadType can be used to determine whether the lookahead priority is included in the message. All lookahead interflow parameters contained in the Route Request message are ATT private data elements.

Example-GetLookAheadPriority
[variable declarations]

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

NOT-QUEUED = 0; LOW = 1; MEDIUM = 2; HIGH = 3; TOP = 4;

NO-IFLOW = -1

HIGH-VDN = 3300
hTelHandle = ICRLAtoi(\$ 1)
lkahdType = GetLookAheadType(hTelHandle)
if lookahead type is not No Interflow, check for

top priority caller

if (lkahdType < > NO-IFLOW) then

if top priority caller, send to VDN that will

queue top priority queue on this switch

if (GetLookAheadPriority(hTelHandle) == TOP)

then
SetPriorityCall(hTelHandle, 1);

SetRouteSelected(hTelHandle,
HIGH-VDN)

RouteFinal(hTelHandle)

endif

endif

Function GetLookAheadSeconds

The GetLookAheadSeconds function returned the lookahead interflow Second. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another conditionally (that is to say the destination switch can either accept or decline the call.) The lookahead second is the second part of the current time when the call was interflowed from the first switch. The lookahead interflow information is filled in by the switch that interflows the call.

THIS PAGE BLANK (USPTO)

Syntax

destVariable = GetLookAheadSeconds(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing the Interflow Minute.
hTelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

Return Value

This function returns an integer value that represents the lookahead interflow second. The lookahead interflow hour will be present if the lookahead interflow type is a valid one. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

The lookahead interflow second is one of six lookahead interflow information components. The remaining five are lookahead type, lookahead priority, lookahead source VDN, lookahead hours, and minutes. The lookahead interflow type can be used to determine whether any of the interflow information exists for a given call.

All lookahead interflow parameters contained in the Route Request message are ATT private data elements.

```
Example-GetLookAheadSeconds
[variable declarations]
NO-IFLOW = -1 .
# obtain telephony handle by converting call ID to
```

numeric value

```
hTelHandle = ICRLAtoi( $ 1 )
lkahdType = GetLookAheadType( hTelHandle )
lkahdHours = GetLookAheadHours( hTelHandle )
lkahdMinutes = GetLookAheadMinutes( hTelHandle )
lkahdSeconds = GetLookAheadSeconds( hTelHandle )
# If lookahead type is not no interflow, output the
```

time of the switch interflow

```
if (lkahdType < > NO-IFLOW) then
```

```
println "Call interflowed at : " + lkahdHours +
```

THIS PAGE BLANK (USPTO)

```
":" + lkahdMinutes + ":" + lkahSeconds
```

```
endif
```

Function GetLookAheadSrcVDN

The GetLookAheadSrcVDN function returns the lookahead interflow source VDN. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another conditionally (that is to say the destination switch can either accept or decline the call.) The lookahead source VDN is included if vector processing determined interflow, and is in the form of the VDN name (not its extension) on the interflow originating switch. The lookahead interflow information is filled in by the switch that interflows the call.

Syntax

destVariable = GetLookAheadSrcVDN(hTelHandle)	
Part	Description
destVariable	Variable that receives the string representing an Interflow determining VDN on the originating switch.
h TelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

Return Value

This function returns a string value that represents the lookahead interflow source VDN name, if one exists. The lookahead interflow source VDN will be present if the lookahead interflow type is Vectoring Interflow. For an invalid telephony handle, the function returns the string "NULL-POINTER".

Remarks

The lookahead interflow source VDN is one of six lookahead interflow information components. The remaining five are lookahead type, lookahead priority, lookahead hours, minutes, and second. The lookahead interflow type can be used to determine whether any of the interflow information exists for a given call.

All lookahead interflow parameters contained in the Route Request message are ATT private data elements.

Example-GetLookAheadSrcVDN

```
[variable declarations]
VECTOR-IFLOW = 2
# obtain telephony handle by converting call ID to
```

THIS PAGE BLANK (00010)

numeric value

```
hTelHandle = ICRLAtoi( $ 1 )
lkahdType = GetLookAheadType( hTelHandle)
# If lookahead type is VECTOR-IFLOW, then get source
```

VDN name

```
if (lkahdType < > NO-IFLOW then
```

```
    lkahdSrcVDN = GetLookAheadSrcVDN(hTelHandle)
```

```
println "Call interflowed from VDN:" + lkahdSrcVDN
endif
```

Function GetLookAheadType

The GetLookAheadType function returns the Interflow Type. The lookahead interflow is a G3 switch feature that routes some of the incoming calls from one switch to another conditionally (that is to say the destination switch can either accept or decline the call.) The lookahead interflow information is filled in by the switch that interflows the call.

Syntax

```
destVariable = GetLookAheadType(hTelHandle)
Part
destVariable      Description
                   Variable that receives the
                   integer representing the
                   Interflow type which can be one
                   of the following:
```

```
1      No Interflow (no interflow
        information present)
```

```
0 All Interflow
1 Threshold Interflow
2 Vectoring Interflow
```

```
h TelHandle      Telephony handle obtained from
                  ICRLAtoi($ 1) function call.
```

THIS PAGE BLANK (USPTO)

Return Value

This function returns an integer value that represents the lookahead interflow type. The lookahead interflow type will always have one of the above specified values. For an invalid telephony handle, the error - 4001 (TELERR-INVALID-TEL-HANDLE) is returned.

Remarks

The lookahead interflow type is one of six lookahead interflow information components. The remaining five are lookahead priority, lookahead source VDN, lookahead hours, minutes, and seconds. The lookahead interflow type can be used to determine whether any of the interflow information exists for a given call.

All lookahead interflow parameters contained in the Route Request message are ATT private data elements.

Example-GotLookAheadType

[variable declarations]

NO-IFLOW = - 1; ALL-IFLOW = 0; THOLD-IFLOW = 1;

VECTOR-IFLOW = 2

obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)

lkahdType = GetLookAheadType(hTelHandle)

If lookahead type is VECTOR- IFLOW (2) we can get

the source VDN name

if (lkahdType = = VECTOR - IFLOW) then

println "The lookahead source VDN name is" +

GetLookAheadSrcVDN(hTelHandle)

endif

THIS PAGE BLANK (USPTO)

Function GetMinute

The GetMinute function returns an integer representing the current minute of the current hour, based upon the most recent call to GetCurrentTime within the script.

Syntax

Part	Description
destVariable	Variable receiving an integer that is the current minute (0-59.)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current minute. For the case of an invalid time handle, the function returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and data information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetMinute

```
{variable declarations}
RETOK = 0
INVALID-HNDL = 0
#Create Time Handle
hTimeHandle = CreateTimeHandle ()
# make sure the time handle is valid before
```

accessing the data

```
if (hTimeHandle < > INVALID-HNDL) then
```

```
# obtain the time data information
GetCurrentTime( hTimeHandle )
println "The current time is: " + GetAscTime(
```

```
hTimeHandle)
```

```
hr = GetHour( hTimeHandle ); min =
```

THIS PAGE BLANK (USPTO)

```
GetMinute (hTimeHandle);

sec = GetSecond(hTimeHandle)
println "From components hh: mm:ss:" + hr + ":"

    + min + ":" + sec

mo = GetMonth(hTimeHandle);

day = GetDayOfMonth(hTimeHandle)

yr = GetYear(hTimeHandle)
println "From components, date is: " + mo + "/"

    + day + "/" + yr

println "Day of week:" + GetDayOfWeek(

    hTimeHandle)

println "Daylight Savings Time (On/Off): " +

    GetDst(hTimeHandle)

println "Day of year" +

    GetDayOfYear(hTimeHandle)

DestroyTimeHandle(hTimeHandle)

endif

Function GetMonth

The GetMonth function returns an integer representing the current month of
the year (1-12) based upon the most recent call to GetCurrentTime within the
script.
```

THIS PAGE BLANK (USPTO)

Syntax

destVariable = GetMonth(hTimeHandle)	
Part	Description
destVariable	Variable receiving an integer that is the current month of the year (1-12.)
h TimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current month of the year. For a case of an invalid time handle, the function returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetMonth

```
[variable declarations]
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle ()
# make sure the time handle is valid before
```

accessing the data

```
if (hTimeHandle < > INVALID - HNDL) then
```

```
# obtain the time data information
GetCurrentTime( hTimeHandle )
println "The current time is: " +
```

```
GetAscTime( hTimeHandle)
```

```
hr = GetHour( hTimeHandle ); min =
```

```
GetMinute (hTimeHandle);
```

```
sec = GetSecond(hTimeHandle);
```

THIS IS A COPY OF THE ORIGINAL

Pat. No. 5870464, *
println "From components hh:mm:ss: " + hr

+ ":" + min + ":" + sec

mo = GetMonth(hTimeHandle);

day = GetDayOfMonth(hTimeHandle)

yr = GetYear(hTimeHandle)
println "From components, date is: " + mo

+ "/" + day + "/" + yr

println "Day of week:" + GetDayOfWeek

(hTimeHandle)

println "Daylight Savings Time (On/Off): "

+ GetDst(hTimeHandle)

println "Day of year" +

GetDayOfYear(hTimeHandle)

DestroyTimeHandle(hTimeHandle)

endif

Function GetNumericFieldValue

The GetNumericFieldValue function returns the value for a specified numeric (integer) type field from the "current" database record. A current database record must be identified using the RunQuery operation before accessing fields within the record.

Syntax

destVariable = GetNumericFieldValue(hDBHandle, fieldName)	
Part	Description

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
If the Query did not return OK, no match was

found, try different KEY

If (query Ret = RETOK) then
Perform a RouteMore to a VDN that queries user to

enter SSN
SetRouteSelected(hTelHandle, QUERY SSN);
SetPriorityCall (hTelHandle 0)
RouteMore (hTelHandle)
#try to locate the record by social security

num collected via Route More

query Ret = RunQuery (hDBHandle, "SSN", "m"

(GetIVRDigits(hTelHandle))

#If the SSN query does not match route to

default hunt group

If (query Ret < > RETOK) then

SetRouteSelected(hTelHandle, FAIL-Rte);

SetPriorityCall(hTelHandle, 0)

rteRet = RouteFinal(hTelHandle)

endif

endif
If query Ret was successful, then we want to set
VALIDATION and route to
VDN that will play message telling customer verification
is complete.

E. If (queryRet = RETOK) then

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
SetStringFieldValue (hDBHandle, "Verification",

"COMPLETE")

nRet = UpdateRecord (h tedChTel handle, VALID-VDN; Set

Priority Call (hTelHandle,0)

rteRte = RouteFinal(hTelHandle)

endif

endif
If (rteRET < > RETOK) then

println "Error In" + \$ 0 + "routing call, Err = " +

rteRet

endif

endif

F. DestroyDBHandle(hDBHandle)

The text below refers to the uppercase letters at the left margin of each block in the diagram.

Block A-Variable Declaration

This block declares the global and local variables. The global variables \$ 1 and \$ 0 are system-defined parameters that exist for every script execution. The variables are accessible by a script through the global declaration: global \$ 1, \$ 0. The variable \$ 1 is a text string containing the call identifier assigned to it by the IIR when the Route Request message is received. This identifier obtains the telephony handle later in the script (the telephony handle is simply a numeric version of this identifier). The variable \$ 0 is the script name. The script can use this variable for output to identify the script related to errors or diagnostics.

Items to note in this block include the declaration of variables to be used as constants (all variables defined with all capitals). These variables appear

THIS PAGE BLANK (USPTO)

in uppercase in the example to make them easy to identify as constants in the script, but uppercase is not required.

Block B-Variable Initialization/Resource Handle Acquisition

This block begins with variable initialization and defines the "constants," which is a good coding practice for maintainability. The constants defined here are used to compare against return values from functions, as well as defining standard route values used throughout the script. Also notice the use of the semicolon between commands, allowing multiple commands to be placed on the same line.

The second half of this block deals with obtaining handles to the database and telephony event information. The telephony event handle is obtained by converting the call identifier (\$ 1) to a numeric value. The database handle is created by invoking CreateDBHandle().

Block C-Validate Resource Handles

This block describes the validation of the resource handles obtained (telephony and database, in this case). The handles are tested against the defined constant INVALID -HNDL, and if either of the handles are invalid, the script attempts to perform a final routing of the call to an extension on the switch which might be an ACD queue for general call handling.

The RouteFinal is preceded by the SetRouteSelected and SetPriorityCall functions, which set mandatory fields in the Route Select message to send to the switch. If these two functions are not called preceding the RouteFinal, the call will not be routed successfully. The SetRouteSelected places the destination extension into the message while the SetPriorityCall places the priority of the Route Select message (0 or 1).

Block D-DB Query/RouteMore

This block displays the database query and the RouteMore function which are used when the initial query fails.

The purpose of the RunQuery call is to determine a match between the customer's ANI (obtained via GetCallingDevice) and the "PhoneNo" field in the database. The result of the operation will be zero (0) if a match is found. In this example, if the return from the query is non-zero (no record matches the caller's ANI), a secondary method verifies the account.

The secondary method attempts to match on social security number, which requires prompting and subsequent DTMF input by the caller. The RouteMore command gives control back to a VDN on the switch, while maintaining position in the current IIR script. By setting the selected route to a VDN containing prompting, digit collection (SetRouteSelected(hTelHandle, QUERY-SSN), and a subsequent adjunct route, you can wait for the second Route Request message and pick up in the IIR script after the RouteMore command.

The second RunQuery operation in this block uses information from the Route Request message as a result of the VDN that prompted for SSN digits. This example uses RouteMore only once, but the number of times that the IIR script can pass control back to a VDN on the switch is unlimited, as long as the VDN

THIS PAGE

to which control is passed performs an adjunct route at some point in its processing.

The second RunQuery looks for a match between the field "SSN" in the database and the digits collected in the QUERY-SSN VDN. If the return from the query this time is not successful, the script routes to the default FAIL-RTE extension.

Block E-Update DB Record/Route Final

This block handles the case where either of the two database queries succeeded performed in Block D above. In this case the customer's record is updated to reflect the verification of the account, and the user is transferred to a VDN that notifies the caller of the verification.

The SetStringFieldValue function sets the Verification field in the database for the customer to COMPLETE. After modifying this field, the record is updated in the database using the UpdateRecord function. If the update is successful, the call is routed to the VDN that plays a message notifying the caller of the successful validation, otherwise the caller is sent to the FAIL-RTE extension for general call handling.

Block F-Release Resource Handles

This block is always executed, just as the creation of the database and telephony handles is in Block B. At this point, all resources allocated in the script must be released. The DestroyDBHandle function releases the database handle. The hTelHandle is automatically released when a RouteFinal operation is encountered.

The remainder of this chapter, Table 16, is a quick reference for scripting commands. The details for the commands are in Appendix C, "Command Reference."

TABLE 16
Command Summary

Function Type	Function Name	Function Prototype	Description
GetRteReq	GetVDN	< string > = GetVDN(hTelHandle)	Returns a string that is the original destination VDN of the call. This field is often considered the DNIS for the call.
	*	*	
	*	*	
GetRteReq	GetCallingDevice	< string > = GetCallingDevice(hTelHandle)	Returns a string that is the call originating device. If outbound and PRI facilities, this is the ANI of the caller.
	*	*	
	*	*	
GetRteReq	GetTrunk	< string > = GetTrunk(hTelHandle)	Returns a string that is the trunk group number from which the call originate.
	*	*	Mutually exclusive with

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

	*	the calling device
	*	field.
	< int > =	Returns an integer that
GetRteReq	GetLookAheadType(represents the
	hTelHandle)	lookahead type for
	*	lookahead interflow.
	*	Examples of lookahead
	*	types are no
	*	interflow, all calls,
	*	threshold and
	*	vectoring interflow.
	< int > =	Returns an integer that
GetRteReq	GetLookAheadPriority(represents the
	hTelHandle)	priority in queue for a
	*	call forwarded via
	*	lookahead interflow (not
	*	in queue, low,
	*	medium, or high.)
	< int > =	Returns an integer that
GetRteReq	GetLookAheadHours(is the hour in
	hTelHandle)	military time when the
	*	origin switch
	*	forwarded the call via
	*	lookahead
	*	interflow.
	< int > =	Returns an integer that
GetRteReq	GetLookAheadMinutes(is the minute of
	hTelHandle)	the hour when the
	*	original switch
	*	forwarded the call via
	*	lookahead
	*	interflow.
	< int > =	Returns an integer that
GetRteReq	GetLookAheadSeconds(is the second of
	hTelHandle)	the minute when the
	*	origin switch
	*	forwarded the call via
	*	lookahead
	*	interflow.
	< int > =	Returns a string that is
GetRteReq	GetLookAheadSrcVDN(the name of the
	hTelHandle)	VDN on the original
	*	switch from which the
	*	call was forwarded via
	*	lookahead
	*	interflow.
	< int > = GetNumIVRSets(Returns an integer that
GetRteReq	hTelHandle)	is the number of
ASISpecif		user collected digit sets

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

ic	*	*	(IVR sets) as a result of multiple RouteMore calls.
	*	*	
GetRteReq	SetCurrentIVRSets	< int > = SetCurrentIVRSets(hTelHandle, IVRSetId)	Returns 0 on successful completion; Parameter IVRSetId is the iteration of user collected digits desired (0 based, 0 is the first route request data.)
ASISpecific	*	*	
ic	*	*	
	*	*	
GetRteReq	GetIVRType	< int > = GetIVRType(hTelHandle)	Returns an integer that is the user collected digits type (none, login digits, database provided, DTMF detector, etc.)
	*	*	
	*	*	
GetRteReq	GetIVRIndicator	< int > = GetIVRIndicator(hTelHandle)	Returns an integer that is the user collected digits indicator (entered or
	*	*	
	\$ er that represents the		
	*	*	type of UII fields (none, user specific-binary, or ASCII text.)
	*	*	Returns an integer that is the length of the UII information. field up to 32 bytes (0 represents no information in the field.)
GetRteReq	GetIncoming UIILength	< int > = GetIncoming UIILength(hTelHandle)	
	*	*	
	*	*	
	*	*	
GetRteReq	GetIncoming UIIData	< string > = GetIncomingUIIData(hTelHandle)	Returns a string that contains the user-to-user information.
	*	*	
SetRteSelect	SetRouteSelected	< int > = SetRouteSelected(hTelHandle, routeSelected)	Returns an integer 0 if successful, otherwise an error on setting route. The parameter routeSelected is a text string containing the route extension to store. REQUIRED field to route call.
	*	*	
	*	*	
	*	*	
SetRteSelect	SetDirectedAgentCall	< int > = SetDirectedAgentCallSplit(hTelHandle, splitId)	Returns an integer 0 if successful, otherwise an error on setting splitId. The
ect	ll		
	Split		

THIS PAGE BLANK (000000)

Pat. No. 5870464, *

		*		parameter Split ID is a
		*		text string
		*		containing the agent
		*		split/skill of the
		*		agent login-id specified
		*		in the
		*		routeSelected.
SetRteSel	SetDestRoute		< int > = SetDestRoute(Returns an integer 0 if
ect			hTelHandle, rteInfo)	successful,
		*		otherwise an error on
		*		setting the
		*		destination route
		*		information. The
		*		parameter retInfo is a
		*		text string
		*		specifying the
		*		TAC/ARS/AAR information
		*		for
		*		off-PBX destinations.
SetRteSel	SetPriorityCall		< int > =	Returns and integer 0 if
ect			SetPriorityCall(successful,
			hTelHandle,	otherwise an error on
		*	priority)	setting the route
		*		priority. The parameter
		*		priority is an
		*		integer that defines
		*		priority for on-PBX
		*		extension (on or off.)
		*		REQUIRED field to
		*		route call.
SetRteSel	SetUserProvidedCode		< int > =	Returns an integer 0 if
ect			SetUserProvidedCode(successful,
		*	hTelHandle,	otherwise an error. The
		*	type, code)	parameter type is
		*		an integer with values
		*		none or database
		*		provided. The parameter
		*		code is a text
		*		field where applications
		*		can set digit
		*		strings that are treated
		*		as dial-ahead
		*		digits on return.
SetRteSel	SetUserCollectCode		< int > =	Returns an integer 0 if
ect			SetUserCollectCode(successful,
		*	hTelHandle, type,	otherwise an error. <
		*	digitsToCollect,	comeback >
		*	timeout, specEvent)	
SetRteSel	SetOutgoingUUI		< int > =	Returns an integer 0 if
ect			SetOutgoingUUI(successful,
		*	hTelHandle), type,	otherwise an error. The
		*	length, UUI)	parameter type is
		*		an integer with possible

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

	*	*	values of none,
	*	*	user specific, or text.
	*	*	The parameter
	*	*	length is an integer set
	*	*	to the length of
	*	*	data, and UUI is a text
	*	*	field containing
	*	*	the UUI.
SetRteSel		< int > =	Returns an integer 0 if
ect	RouteFinal	RouteFinal(hTelHandle)	successful,
	*	*	otherwise an error from
	*	*	the route request.
	*	*	This function causes the
	*	*	Route Select
	*	*	message to be e
	*	*	adjunct step being
	*	*	executed at the switch.
		< int > =	Returns an integer 0 if
GetAgents		QueryAgentState(successful. The
tate	QueryAgentState	hTelHandle,	parameter agDevice is a
	*	agDevice, agSplit)	text string that
	*	*	identifies the agent
	*	*	login-id. The
	*	*	parameter agSplit is a
	*	*	text string that
	*	*	identifies one of the
	*	*	valid splits/skills
	*	*	associated with the
	*	*	agent.
		< int > =	Returns an integer that
GetAgents		GetAgentAvailable(is the agent
tate	GetAgentAvailable	hTelHandle)	available state (either
ASISpecif	*	*	true or false.)
ic	*	*	This is an ASI specific
	*	*	combination of
	*	*	agentState and agent talk
	*	*	state. When
	*	*	true the agent is logged
	*	*	in and is ready
	*	*	to accept a call.
GetAgents		< int > = GetAgentState(Returns an integer that
tate	GetAgentState	hTelHandle)	represents the
	*	*	agent state (based upon
	*	*	the
	*	*	QueryAgentState.)
	*	*	Possible agent state
	*	*	include not ready, null,
	*	*	ready, work not
	*	*	ready, and work ready.
		< int > =	Returns an integer that
GetAgents		GetAgentWorkMode(represents the
tate	GetAgentWorkMode	hTelHandle)	agent work mode (based

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

	*	*	upon the
	*	*	QueryAgentState.)
	*	*	Possible agent work
	*	*	modes include aux work,
	*	*	after call work,
	*	*	auto in, and manual in.
GetAgents		< int > =	Returns an integer that
tate	GetAgentTalkState	GetAgentTalkState(represents the
		hTelHandle)	agent talk (based upon
	*	*	the
	*	*	QueryAgentState.)
	*	*	Possible agent states
	*	*	are on call, and idle
Database	CreateDBHandle	dbHandle =	Returns a numeric DB
		CreateDBHandle()	handle for this
	*	*	script instance. This
	*	*	dbHandle will be
	*	*	used throughout the IIR
	*	*	script instance to
	*	*	access the DB. If unable
	*	*	to allocate DB
	*	*	access, a NULL (0) will
	*	*	be returned.
Database	DestroyDBHandle	< int > =	Returns a 0 if
		DestroyDBHandle(successful. This function
		dbHandle)	must be called before
	*	*	exiting an IIR
	*	*	script to release the
	*	*	database resources
	*	*	associated with the
	*	*	dbHandle allocated
	*	*	with CreateHandle().
Database	RunQuery	< int > =	Returns 0 if successful,
		RunQuery(dbHandle,	otherwise the
		fieldName,	error associated with the
	*	queryOp, keyValue)	DB query.
	*	*	Parameter fieldName is
	*	*	the actual field
	*	*	name defined in the DB
	*	*	admin, queryOp is
	*	*	the operator for
	*	*	comparison (i.e. =, <
	*	*	, >, etc.), and keyValue is
	*	*	the key value for
	*	*	the query.
Database	InsertRecord	< int > =	Returns 0 if successful,
		InsertRecord(dbHandle)	otherwise the
	*	*	error on the insert of
	*	*	the record. Record
	*	*	is created by using
	*	*	SetStringFieldValue

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

and SetNumericFieldValue.
The record
inserted is an inherent
part of the
dbHandle.
Returns 0 if successful,
otherwise the
error on the update of
the record. Record
is updated by using
SeStringValue < int
> = MovePreviousRecord(
dbHandle)

Database UpdateRecord

< int > =
UpdateRecord(dbHandle)

Returns 0
if
successful,
otherwise
an

error. Moves to the
previous record in
the direction of search
in the database,
based upon the key search
value in
RunQuery.

Database

GetStringFieldValu
e GetStringFieldValue(
dbHandle,
fieldName)

Returns 0 if successful,
otherwise an
error. Returns the string
value for the
specified fieldName in
the current
record.

Database

GetNumericFieldVal
ue GetNumericFieldValue(
dbHandle,
fieldName)

Returns 0 if successful,
otherwise an
error. Returns the
numeric value for the
specified fieldName in
the current record.

Database

SetStringValue
e SetStringValue(
dbHandle,
fieldName, setValue)

Returns 0 if successful,
otherwise an
error. Sets the fieldName
in the current
record to the text string
in setValue.

Database

SetNumericFieldVal
ue SetNumericFieldValue(
dbHandle,
fieldName, setValue)

Returns 0 if successful,
otherwise an
error. Sets the fieldName
in the current
record to the integer in
setValue.

timeHandle =

Returns a non-zero

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

MiscTime	CreateTimeHandle	CreateTimeHandle()	integer if successful, otherwise no handle was allocated. Must be performed only once per script.
	*	*	
	*	*	
MiscTime	DestroyTimeHandle	< int > = DestroyTimeHandle(timeHandle)	Returns 0 if successful. Destroys the time handle allocated with CreateHandle(). TimeHandle must be destroyed before exiting the IIR script.
	*	*	
	*	*	
	*	*	
MiscTime	GetCurrentTime	< int > = GetCurrentTime(timeHandle)	Returns 0 if successful. Obtains the current time associated with the timeHandle.
	*	*	
	*	*	
MiscTime	GetAscTime	< string > = GetASCTime(timeHandle)	Returns a string containing the date and time in string format.
	*	*	
MiscTime	GetHour	< int > = GetHour(timeHandle)	Returns an integer as the hour in military time based upon the most recent GetCurrentTime.
	*	*	
	*	*	
MiscTime	GetMinute	< int > = GetMinute(timeHandle)	Returns an integer as the minute of the hour based upon the most recent GetCurrentTime.
	*	*	
	*	*	
MiscTime	GetSecond	< int > = GetSecond(timeHandle)	Returns an integer as the second of the minute based upon the most recent GetCurrent Time.
	*	*	
	*	*	
MiscTime	GetDayOfMonth	< int > = GetDayOfMonth(timeHandle)	Returns an integer as the numeric day of month based upon the most recent GetCurrent Time.
	*	*	
	*	*	
MiscTime	GetMonth	< int > = GetMonth(timeHandle)	Returns an integer representing the month based upon the most recent GetCurrent Time.
	*	*	
	*	*	
MiscTime	GetYear	< int > = GetYear(timeHandle)	Returns an integer (last two digits) of the current year based upon the most recent GetCurrent Time.
	*	*	
	*	*	
MiscTime	GetDayOfWeek	< int > = GetDayOfWeek(timeHandle)	Returns an integer representing the day of length of the string

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

	*	*	parameter.
ystem	ICRLStrStr	< string > = ICRLStrStr(string, findStr)	Returns a string starting with the findStr in the string parameter passed in.
	*	*	
ystem	ICRLStrIndex	< int > = ICRLStrIndex(string, findStr, startIdx)	Returns an integer index where findStr is located within string. The search is started at the startIdx in the string parameter.
	*	*	
	*	*	
ystem	ICRLLeft	< string > = ICRLLeft(string, numChars)	Returns the substring that is the left most numChars of the string parameter.
	*	*	
System	ICRLRight	< string > = ICRLRight(string, numChars)	Returns the substring that is the right most numChars of the string parameter.
	*	*	
System	ICRLMid	< string > = ICRLMid(string, startChar, numChars)	Returns the substring defined to start at startChar (0 based), and of numChars length of the string parameter.
	*	*	
System	ICRLatoi	< int > = ICRLatoi(string)	Return the integer value for the string parameter.
	*	*	
System	ICRLStrCopy	< string > = ICRLStrCopy(string)	Makes a copy of the specified string parameter.
	*	*	

Appendix C, "Command Reference," provides a detailed description with examples for each of these commands.

APPENDIX A-COMMAND REFERENCE

The commands in this section are presented in alphabetical order. Each command begins on a new page to make reference easier. For a quick summary only, see the previous section.

Function ClearRecord

The ClearRecord function removes a record in the database that has been located through the RunQuery or MoveNextRecord/MovePreviousRecord functions.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

Syntax

destVariable = ClearRecord(hDBHandle)	
Part	Description
destVariable	Variable receiving integer value representing success/failure of call.
hDBHandle	Variable that receives the allocated database handle

Return Value

Returns 0 if successful, for an invalid database handle the function returns 5008. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve are covered in the IIR trouble shooting guide.

Remarks

The ClearRecord function cannot be called before a database handle being obtained through the CreateDBHandle function, and a "current" record being defined through the RunQuery function call.

Note that after the record has been cleared, the current database record associated with the hDBHandle is undefined, and therefore any access to record data before another RunQuery call will result in an error.

Example-ClearRecord

```
[variable declarations]
INVALID-HNDL = 0; RETOK = 0
hDBHandle = CreateDBHandle( )
hTelHandle = ICRLAtol($ 1)
# If unable to create a database handle, do default
route processing
if (hDBHandle == INVALID-HNDL) then
perform default route processing
else
nRet = RunQuery(hDBHandle, "AccountNo", " = ",
GetIVRDigits( hTelHandle))
# If the Query returned return OK, then delete the
record from the database
if (nRet == RETOK ) then
if (ClearRecord( hDBHandle ) < > RETOK ) then
process error on deleting record
endif
endif
```

```
endif
DestroyDBHandle( hDBHandle )
```

THIS PAGE BLANK (USPTO)

Function CreateDBHandle

The CreateDBHandle function allocates a database handle from the resource manager, and provides the means to access and update database related information through the database related functions.

Syntax

hDBHandle = CreateDBHandle()	Description
Part	Variable that receives the
hDBHandle	allocated database handle

Return Value

On failure, this function returns an integer value of 0 if the resource manager is unable to allocate a database handle.

Remarks

The CreateDBHandle function provides the access mechanism for all database related functions. The CreateDBHandle function is closely related to another of the database functions, DestroyDBHandle.

The DestroyDBHandle function must be called before exiting the IIR script to ensure proper management of the database handle resources. The allocation of a database handle does not locate a record within the database. Before performing any database record reads, updates, or deletions, the RunQuery function must be called successfully.

Example-CreatedDBHandle

```
[variable declarations]
INVALID-HNDL = 0
hDBHandle = CreateDBHandle( )
hTelHandle = ICRLAtoi($ 1)
# If unable to create a database handle, do default
route processing
if (hDBHandle == INVALID-HNDL) then
perform default route processing
else
nRet = RunQuery( hDBHandle, "AccountNo", " = ",
GetIVRDigits( hTelHandle ) )
# If the Query returned return OK, then print
out the customer information
if ( nRet == RETOK) then
lstName = GetStringFieldValue(hDBHandle,
"LastName")
frstName = GetStringFieldValue(hDBHandle,
"FirstName")
delCount = GetNumericFieldValue(hDBHandle,
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
"DelinqCount")
process caller based upon delinquent count
and log if delinquent > 3
if (delcount > 3) then
println "Customer" + 1stName + ", " + frstName + "
has del = " + delCount
endif
endif
```

```
endif
DestroyDBHandle( hDBHandle)
```

Function Create Time Handle

The CreateTimeHandle function allocates a time handle from the resource manager, and provides the means to access time/date related information through the Get time/date related functions.

Syntax

hTimeHandle = CreateTimeHandle()	Description
Part	Variable that receives the time
hTimeHandle	handle

Return Value

On failure, this function returns a value of 0 if the resource manager cannot allocate a time handle.

Remarks

The CreateTimeHandle function provides the access mechanism for all time and date related functions such as GetHour, GetDayOfMonth, etc. The CreateTimeHandle function is closely related to two of the other time resource functions, DestroyTimeHandle, and GetCurrentTime.

The DestroyTimeHandle function must be called before exiting the IIR script to ensure proper management of the time handle resources. A call to GetCurrentTime after obtaining the time handle is necessary to obtain/update the time and date related information. All of the time/date Get . . . functions obtain their information based upon the most recent call to GetCurrentTime.

Example-CreateTimeHandle

```
{variable declarations}
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle( )
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

make sure the time handle is valid before

accessing the data

if (hTimeHandle < > INVALID-HNDL) then

```
# obtain the time data information
GetCurrentTime( hTimeHandle )
println "The current time is: " + GetAscTime(
hTimeHandle)
hr = GetHour( hTimeHandle ); min =
GetMinute(hTimeHandle);
sec = GetSecond(hTimeHandle)
println "From components hh:min:ss: " + hr + ":"
+ min + ":" + sec
mo = GetMonth(hTimeHandle);
day = GetDayOfMonth(hTimeHandle)
yr = GetYear(hTimeHandle)
println "From components, date is: " + mo + "/"
+ day + "/" + yr
println "Day of week:" + GetDayOfWeek
(hTimeHandle)
println "Daylight Savings Time (On/Off):" +
GetDst(hTimeHandle)
println "Day of year:" +
GetDayOfYear(hTimeHandle)
DestroyTimeHandle( hTimeHandle )
```

endif

Function DestroyDBHandle

The DestroyDBHandle function frees a database handle back to the resource manager. This function is paired with the CreateDBHandle function, which allocates the database handle at the beginning of an IIR script.

Syntax

destVariable = DestroyDBHandle(hDBHandle)	
Part	Description
destVariable	Variable receiving integer value representing success/failure of call
hDBHandle	Parameter that is the database handle in the script

THIS PAGE BLANK (USPTO)

Return Value

If successful, this function returns 0. For an invalid database handle, this function returns an integer value of - 5008.

Remarks

The DestroyDBHandle function must be called before exiting the IIR script to ensure proper management of the database handle resources, if a database handle has been allocated using the CreatedBHandle function before that point in the script.

```
Example-DestroyDBHandle
{variable declarations}
INVALID-HNDL = 0
hDBHandle = CreatedBHandle( )
hTelHandle = ICRLAtoi($ 1)
# If unable to create a database handle, do default
```

route processing

```
if ( hDBHandle == INVALID-HNDL) then
```

perform default route processing

else

```
nRet = RunQuery( hDBHandle, "AccountNo", " = ",
GetIVRDigits( hTelHandle) )
# If the Query returned return OK, then print
out the customer information
if ( nRet == RETOK) then
  lstName = GetStringFieldValue( hDBHandle,
  "LastName")
  frstName = GetStringFieldValue (hDBHandle,
  "FirstName")
  delCount = GetNumericFieldValue (hDBHandle,
  "DelinqCount")
  process caller based upon delinquent count
  and log if delinquent > 3
  if (delcount > 3) then
    println "Customer" + lstName + ",
    " + frstName + " has del =
    " + delCount
  endif
endif
```

```
endif
DestroyDBHandle( hDBHandle)
```

THIS PAGE BLANK (USPTO)

Function DestroyTime Handle

The DestroyTimeHandle function frees a time handle back to the resource manager. This function is paired with the CreateTimeHandle function which allocates the time handle at the beginning of an IIR script.

Syntax

Part	Description
destVariable	Variable receiving integer value representing success/failure of call.
hTimeHandle	Variable that receives the time handle

Return Value

This function returns a value of 0 if successful; - 5008 if invalid.

Remarks

The DestroyTimeHandle function must be called before you exit the IIR script to ensure proper management of the time handle resources if a time handle has been allocated using the CreateTimeHandle function before that point in the script.

```
Example-DestroyTimeHandle
[variable declarations]
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle( )
# make sure the time handle is valid before
```

accessing the data

```
if (hTimeHandle < > INVALID-HNDL) then
# obtain the time data information
```

```
GetCurrentTime( hTimeHandle)
println "The current time is: " + GetAscTime(
hTimeHandle)
hr = GetHour( hTimeHandle) min =
GetMinute(hTimeHandle);
sec = GetSecond(hTimeHandle)
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
println "From components hh:mm:ss: " + hr + ":"
+ min + ":" + sec
mo = GetMonth(hTimeHandle);
day = GetDayOfMonth(hTimeHandle)
yr = GetYear(hTimeHandle)
println "From components, date is: " + mo + "/"
+ day + "/" + yr
println "Day of week: " + GetDayOfWeek(
hTimeHandle)
println "Daylight Savings Time (On/Off) : " +
GetDst(hTimeHandle)
println "Day of year:" +
GetDayOfYear(hTimeHandle)
nRet = DestroyTimeHandle( hTimeHandle )
```

endif

Function GetAgent Available

The GetAgentAvailable function returns either true or false (1,0) depending upon the state of the agent information obtained through the QueryAgentState call. The agent available state is a combination of the agent state and the agent talk state where the agent is ready and the talk state is idle.

Syntax

destVariable = GetAgentAvailable(hTelHandle)	
Part	Description
destVariable	Variable that receives the state of the agent availability.
hTel/Handle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.

Return Value

This function returns an integer value that represents the available state of the agent defined in a previously executed QueryAgentState call. If the telephony handle is invalid, this function returns - 4001.

Remarks

This function is dependent upon the agent information obtained in the QueryAgentState function call. This function will return an error if the QueryAgentState function has not been called with to obtain the agent state information for the specific agent.

Example-GetAgentAvailable

```
[variable declaration]
SKILL -HUNT = "2800"
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
hTelHandle = ICRLAtoi (\$ 1)
"3600" represents a specific agents login-ID
nRet = QueryAgentState(hTelHandle, "3600",

SKILL-HUNT)

if (GetAgentAvailable(hTelHandle)) then

send caller to this specific agent

else

send caller to general queue for handling

endif

Function GetAgentState

The GetAgentState function returns the "ready" state of a given agent based upon the agent information obtained through the QueryAgentState call. This represents the agentState field from the Query Agent State Service message.

Syntax

destVariable = GetAgentState(hTelHandle)
Part Description
destVariable Variable that receives the agent
"ready" state, possible values
are as follows:

0 = AG-NOT-READY - Agent is not ready to

receive calls

1 = AG-NULL - Agent is not logged in on

specified device/split

2 = AG-READY - Agent is available for calls

or talking

THIS PAGE BLANK (USPTO)

3 = AG-WORK-NOT-READY - After call work

occupied

hTelHandle Telephony handle obtained from
ICRLAtoi(\$ 1) function call.

Return Value

This function returns an integer value that represents the agent "ready" state base upon the most recently executed QueryAgentState call. If the telephony handle is invalid, this function returns - 4001.

Remarks

This function is dependent upon the agent information obtained in the QueryAgentState function call. This function will return an error if the QueryAgentState function has not been called with to obtain the agent state information for the specific agent.

Example-GetAgentState

```
[variable declarations]
SKILL-HUNT = "2800"; AG-READY = 2; TS-IDLE = 1
hTelHandle = ICRLAtoi ($ 1)
"3600" represents a specific agents login-ID
= QueryAgentState( hTelHandle, "3600", SKILL-HUNT)
agState = GetAgentState( hTelhandle )
agTalkState = GetAgentTalkState( hTelHandle )
if ( agState == AG-READY And agTalkState == TS-IDLE )
then
```

send caller to this specific agent

else

send caller to general queue for handling

endif

Function GetAgentTalkState

THIS PAGE BLANK (USPTO)

The GetAgentTalkState function returns the talk state of an agent based upon the agent information obtained through the QueryAgentState call. This represents the talkState field from the Query Agent State Service message.

Syntax

```
destVariable = GetAgentState (hTelHandle)
Part          Description
destVariable  Variable that receives the agent
               talk state, possible values are
               as follows:
```

```
0 = TS-ON-CALL - Agent is currently talking
to a caller
1 = TS-IDLE - Agent is waiting for a caller
```

```
hTelHandle    Telephony handle obtained from
               ICRLAtoi($ 1) function call.
```

Return Value

This function returns an integer value that represents the agent talk state base upon the most recently executed QueryAgentState call. If the telephony handle is invalid, this function returns - 4001.

Remarks

This function is dependent upon the agent information obtained in the QueryAgentState function call. This function will return an error if the QueryAgentState function has not been called with to obtain the agent state information for the specific agent.

The agents talkState is only defined when the agentState is AG-READY. See GetAgentState for details on agentState.

Example-GetAgentTalkState
[variable declarations]

```
SKILL-HUNT = "2800"; AG-READY = 2; TS-IDLE = 1
hTelHandle = ICRLAtoi ($ 1)
# "3600" represents a specific agents login-ID
nRet = QueryAgentState( hTelHandle, "3600",
```

```
SKILL-HUNT)
```

```
agState = GetAgentState( hTelhandle )
agTalkState = GetAgentTalkState( hTelHandle)
```

THIS PAGE IS

Pat. No. 5870464, *

```
if ( agState == AG-READY And agTalkState ==
```

```
    TS-IDLE ) then  
    send caller to this specific agent
```

```
else
```

```
    send caller to general queue for handling
```

```
endif
```

Function GetAgentWorkMode

The GetAgentWorkMode function returns the work mode of a given agent based upon the agent information obtained through the QueryAgentState call. This represents the workMode field from the Query Agent State Service message.

Syntax

```
destVariable = GetAgentWorkMode(hTelHandle)  
Part          Description
```

destVariable	Variable that receives the agent work mode, possible values are as follows:
--------------	---

- 1 = WM-AUX-WORK
- 2 = WM-AFTCAL-WK
- 3 = WM-AUTO-IN
- 4 = WM-MANUAL-IN

hTelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.
------------	--

Return Value

This function returns an integer value that represents the agent work mode based upon the most recently executed QueryAgentState call. If the telephony handle is invalid, this function returns - 4001.

Remarks

This function is dependent upon the agent information obtained in the QueryAgentState function call. This function will return an error if the

THIS PAGE BLANK (USPTO)

QueryAgentState function has not been called with to obtain the agent state information for the specific agent.

Example-GetAgentWorkMode

```
[variable declarations]
SKILL-HUNT = "2800"; AG - READY = 2; TS-IDLE = 1
hTelHandle = ICRLAtoi($ 1)
# "3600" represents a specific agents login-I D
nRet = QueryAgentState( hTelHandle, "3600",

                                SKILL-HUNT)

agState = GetAgentState( hTelhandle)
agTalkState = GetAgentTalkState( hTelHandle )
wkMode = GetAgentWorkMode( hTelHandle )
if ( agState == AG-READY And agTalkState == TS-IDLE )

    then
        send caller to this specific agent

else

    send caller to general queue for handling

endif
```

Function GetAscTime

The GetAscTime function returns a string containing the current date and time in the format "Thu Oct. 12 09:00:40 1995" based upon the date and time of the last call to GetCurrentTime in the IIR script.

Syntax

```
destVariable = GetAscTime(hTimehandle)
Part          Description
destVariable   Variable receiving string that is
                ASCII date and time. The data is
                returned in the format "Thu Oct
                12 09:00:40 1995".
hTimeHandle    Variable that was assigned a time
                handle via CreateTimeHandle call.
```

THIS PAGE BLANK (USPTO)

Return Value

If successful, this function returns the string containing the formatted time and date. If unsuccessful, a zero length string will be returned (NULL string.)

Remarks

Allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetAscTime

```
[variable declarations]
```

```
RETOK = 0
```

```
INVALID-HNDL = 0
```

```
# Create Time Handle
```

```
hTimeHandle = CreateTimeHandle()
```

```
# make sure the time handle is valid before
```

```
accessing the data
```

```
if (hTimeHandle < > INVALID HNDL) then
```

```
# obtain the time data information
```

```
GetCurrentTime( hTimeHandle )
```

```
println "The current time is : " + GetAscTime(
```

```
hTimeHandle )
```

```
hr = GetHour(hTimeHandle); min =
```

```
GetMinute(hTimeHandle);
```

```
sec = GetSecond(hTimeHandle)
```

```
println "From components hh:mm:ss " + hr + ":" +
```

```
min + ":" + sec
```

```
mo = GetMonth(hTimeHandle);
```

```
day = GetDayOfMonth(hTimeHandle)
```

```
yr = GetYear(hTimeHandle).
```

THIS PAGE BLANK (00710)

Pat. No. 5870464, *
println "From components, date is: " + mo + "/"

+ day + "/" + yr

println "Day of week:" + GetDayOfWeek(

hTimeHandle)

println "Daylight Savings Time (On/Off): " +

GetDst(hTimeHandle)

println "Day of Year:" +

GetDayOfYear(hTimeHandle)

DestroyTimeHandle:(hTimeHandle)

endif

Function GetCallingDevice

The GetCallingDevice function returns the calling device number (extension or phone number) of the originating party. For calls that originate on-PBX, or incoming calls over PRI facilities, the calling device number is returned. No information is returned by this command for incoming calls over non-PRI lines, however, the Trunk Group Number is available from the GetTrunk command. This parameter is obtained from the Route Request Service message (Version 2), callingDevice field.

Syntax

destVariable = GetCallingDevice(hTelHandle)

Part	Description
destVariable	Variable that receives the string returned from this function.
hTelHandle	Telephony handle obtained from ICRLAtoi(\$ 1) function call.

Return Value

THIS PAGE BLANK (USPTO)

This function returns a string expression that represents the calling device extension/number. If the return value is a zero length string, no callingDevice information was received for this call. If the telephony handle is not valid, this function returns the string "NULL-POINTER".

Remarks

The callingDevice field in the Route Request Service message is an optional component of this message, therefore it may not exist for each call. This field, for calls incoming over PRI facilities is the ANI (Automatic Number Identification) value. The callingDevice field is mutually exclusive with the trunk field accessed through the GetTrunk function. This means one or the other of callingDevice or trunk are available in the message, but never both.

Example-GetCallingDevice

```
{variable declarations}
# obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi( $ 1 )
ANI = GetCallingDevice( hTelHandle )
# checking for a zero length string determines

whether the field is in the message

if (ANI < > "") then

    println "The calling device (ANI) is = " + ANI

else

    # the trunk data will be available when the

        calling device isn't

        trunkID = GetTrunk( hTelHandle )
        println "The inbound trunk group ID = " +

            trunkID

endif
```

THIS PAGE BLANK (b)(7)(D)

Function GetCurrentTime

The GetCurrentTime function sets or refreshes the current date and time associated with the specified time handle. All Get . . . time and date functions receive information based upon the last invocation of this function.

Syntax

Part	Description
destVariable	Variable receiving integer specifying success/failure of call.
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns 0; if invalid time handle, this function returns a value of - 5008.

Remarks

A call to GetCurrentTime after obtaining the time handle is necessary to obtain/update the time and date related information. All of the time/date Get . . . functions obtain their information based upon the most recent call to GetCurrentTime. Allocate the time handle by calling CreateTimeHandle before calling this function.

Example-GetCurrentTime

```
[variable declarations]
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle()
# make sure the time handle is valid before
```

accessing the data

```
if (hTimeHandle < > INVALID-HNDL) then
```

```
# obtain the time data information
GetCurrentTime( hTimeHandle)
println "The current time is:" +
```

```
GetAscTime( hTimeHandle)
```

THIS PAGE BLANK (USPTO)


```
hr = GetHour( hTimeHandle ); min =  
  
    GetMinute(hTimeHandle);  
  
sec = GetSecond(hTimeHandle)  
println "From components hh:mm:ss " + hr +  
  
    ":" + min + ":" + sec  
  
mo = GetMonth(hTimeHandle);  
  
    day = GetDayOfMonth(hTimeHandle)  
  
yr = Getyear(hTimeHandle)  
println "From components, date is: " + mo +  
  
    "/" + day + "/" + yr  
  
println "Day of week:" + GetDayOfWeek(  
  
    hTimeHandle)  
  
println "Daylight Savings Time (On/Off) : "  
  
    + GetDst(hTimeHandle)  
  
println "Day of year:" +  
  
    GetDayOfYear(hTimeHandle)  
  
DestroyTimeHandle( hTimeHandle)  
  
endif
```

Function GetDayOfMonth

THIS PAGE BLANK (USPTO)

The GetDayOfMonth function returns an integer representing the current day of the month (1-31) based upon the most recent call to GetCurrentTime within the script.

Syntax

destVariable = GetDayOfMonth(hTimeHandle)	
Part	Description
destVariable	Variable receiving an integer that is the current day of the month (1 -31.)
hTimeHandle	Variable that was assigned a time handle via CreateTimeHandle call.

Return Value

If successful, this function returns an integer defining the current day of the month. If invalid, the function returns - 5008.

Remarks

It is necessary to allocate the time handle by calling CreateTimeHandle and obtain the time and date information through a call to GetCurrentTime before calling this function. The information returned by this call will be based upon the last call to GetCurrentTime in the script.

Example-GetDayOfMonth

```
[variable declarations]
RETOK = 0
INVALID-HNDL = 0
# Create Time Handle
hTimeHandle = CreateTimeHandle()
# make sure the time handle is valid before

    accessing the data

if (hTimeHandle < > INVALID-HNDL) then

    # obtain the time data information
    GetCurrentTime( hTimeHandle )
    println "The current time is:" + GetAscTime(

        hTimeHandle)

    hr = GetHour( hTimeHandle ); min =
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
GetMinute(hTimeHandle);

sec = GetSecond(hTimeHandle)
println "From components hh:mm:ss: " + hr + ":"

+ min + ":" + sec

mo = GetMonth(hTimeHandle);

day = GetDayOfMonth(hTimeHandle)

yr = GetYear(hTimeHandle)
println "From components, date is:" + mo + "/"

+ day + "/" + yr

println "Day of week:" + GetDayOfWeek

(hTimeHandle)

println "Daylight Savings Time (On/Off): " +

GetDst(hTimeHandle)

println "Day of year:" +

GetDayOfYear(hTimeHandle)

DestroyTimeHandle:(hTimeHandle)

endif

Function GetDayOfWeek

The GetDayOfWeek function returns an integer representing the current day of the week (1-7); the base or first day (1) being Sunday, based upon the most recent call to GetCurrentTime within the script.

PAGE BLANK (USPTO)

The RouteFinal function call must be preceded by the setting of the mandatory route fields priorityCalling and routeSelected through the functions SetPriorityCall and SetRouteSelected respectively. If these fields are not defined before the RouteFinal is attempted, the G3 Telephony Services driver will reject the RouteSelect message and will result in the call not being routed.

When this occurs, the VDN on the switch which has transferred control to the IIR Script will timeout waiting for a response from the adjunct request, and will subsequently execute the next command in the vector processing the that call.

```
Example-RouteFinal
[variable declarations]
DFLT- RTE = "3100"
hTelHandle = ICRLAtoi ( $ 1)
# if the VDN is equal to accounting DNIS, route to
```

```
acct hunt group extension
```

```
if (GetVDN( hTelHandle ) == "2222") then
```

```
    DFLT-RTE = "3300"
```

```
endif
SetRouteSelected( hTelHandle, DFLT-RTE)
SetPriorityCall( hTelHandle, 0)
RouteFinal( hTelHandle)
```

Function RouteMore

The RouteMore function is one of three route specification functions (RouteMore, RouteFinal, and RouteUnknown.) This function requests that call control be routed back to an intermediate VDN for collection of more user entered digits, while maintaining control of the call in the IIR script.

Syntax

```
destVariable = RouteMore (hTelHandle)
```

Part	Description
destVariable	Variable that receives the integer representing success/failure of call.
h TelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.

THIS PAGE BLANK (USPTO)

Return Value

If successful the function returns 0, for an invalid telephony handle - 4001, and for a timeout waiting for the RouteRequest message, the function returns error - 208.

Remarks

There are three functions related to the IVR set concept, RouteMore, GetNumIVRSets, and SetCurrentIVRSets. The RouteMore function allows an IIR script to perform multiple IVR digit queries with the caller without leaving the control scope of the one IIR script.

The GetNumIVRSets allows the script to obtain the number of IVR digit sets that are potentially available, based upon the first script invocation as well as all RouteMore calls for more collected digits. The RouteMore call actually sends control back to a call prompting VDN to obtain more digits, where control is returned to the IIR script through the call vectoring adjunct command. Upon return from the RouteMore function, the most recent set of IVR collected digits is automatically available through the GetIVR functions (GetIVRType, GetIVRIndicator, and GetIVRDigits.)

The script can, however, also access any previous "set" or IVR data (within the scope of this one IIR script) by forcing the previous "set" of IVR data to focus with the SetCurrentIVRSets function.

When the RouteMore function is used, the script will continue executing at the command following the RouteMore call when the intermediate VDN completes its call vectoring task and issues an adjunct route request.

The RouteMore function call must be preceded by the setting of the mandatory route fields priorityCalling and routeSelected through the calls SetPriorityCall and SetRouteSelected respectively.

Additionally, the SetRouteSelected function should identify a VDN having digit collection capabilities and most importantly an adjunct route command in the body of the associated call vector on the G3 switch. This adjunct route command assures that control will be returned to the awaiting IIR script.

If control is not returned within the default time out period, the IIR script will automatically terminate based upon the "RouteMoreTimer-nn" entry in the CTI.CFG file found in the installation directory of the NetWare NLMs.

Example-RouteMore

[variable declarations]

NO-IVRDATA = - 1; GET-PIN-VDN = "3200"; FIRST-SET = 0;

BOTH-SETS = 2

obtain telephony handle by converting call ID to

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

numeric value

```
hTelHandle = ICRLAtoi ( $ 1 )
} check to see if account number was entered by caller,
```

if so collect PIN number

```
if (GetIVRType( hTelHandle ) < > NO - IVRDATA) then
```

```
SetPriorityCall( hTelHandle, 0); SetRouteSelected
```

```
(hTelHandle, GET-PIN-VDN)
```

```
RouteMore( hTelHandle )
if (GetNumIVRSets(hTelHandle) == BOTH-SETS) then
```

```
pinNum = GetIVRDigits( hTelHandle )
SetCurrentIVRSets( hTelHandle, FIRST - SET)
acctNum = GetIVRDigits( hTelHandle )
Perform database match on acctNum/pinNum and
```

process call

endif

endif

Function RouteUnknown

The RouteUnknown function is one of three route specification functions (RouteMore, RouteFinal, and RouteUnknown). An IIR script uses this function to route a call for the last time in the context of the current script when there is no known route as determined by the script. The result is execution of the command immediately following the adjunct route command in the VDN requesting route direction.

Syntax

destVariable = RouteUnknown(hTelHandle)	
Part	Description
destVariable	Variable that receives the integer representing success/failure of call. Any

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
non-zero failure code received
can include any of the error
codes defined for the telephony
functions in the trouble-shooting
guide.
Telephony handle obtained from
ICRLAtoi (\$ 1) function call.

h TelHandle

Return Value

If successful the function returns 0, for an invalid telephony handle - 4001.
All other non-zero error codes can be referenced to the telephony function error
codes in the trouble-shooting guide.

Remarks

Performing the RouteUnknown operation implies that no FINAL destination for
the call has been determined for the execution of this script. The RouteUnknown
command inherently releases the telephony handle obtained from the call to
ICRLAtoi(\$ 1). Any reference to the telephony handle after the RouteUnknown call
in the script will result in an invalid telephony handle error being returned.

The RouteUnknown function, unlike the RouteFinal and RouteMore commands, does
NOT require that the priorityCalling and routeSelected fields to be defined
before invocation. If the route is unknown, these fields would obviously be
meaningless in the message. When this occurs, the VDN on the switch which has
transferred control to the IIR Script will timeout waiting for a response from
the adjunct request, and will subsequently execute the next command in the
vector processing for that call.

Example-RouteUnknown
[variable declarations]
hTelHandle = ICRLAtoi (\$ 1)
if the VDN is equal to accounting or marketing

DNIS, route to default hunt group extension

if (GetVDN(hTelHandle) == "2222" Or
GetVDN(hTelHandle) == "3333") then

SetRouteSelected(hTelHandle, DFLT-RTE);

SetPriorityCall(hTelHandle, 0)

RouteFinal(hTelHandle)

else

THIS PAGE BLANK (USPTO)

```
# no valid route is known for the call, leave
```

```
handling of call to originating VDN
```

```
RouteUnknown( hTelHandle)
```

```
endif
```

Function Run Query

The RunQuery function is used to query the IIR database and locate a matching record in the database. Based upon the query criteria, a matching database record may or may not be found. If the RunQuery function returns successfully, and hDBHandle can be used in combination with other database functions to read, update, or delete the current record.

Syntax

```
destVariable = RunQuery(hDBHandle, fieldName, queryOp, keyValue)
```

Part	Description
destVariable	Variable receiving integer value representing success/failure of call.
hDBHandle	Parameter that is the database handle in the script
fieldName	Parameter of string type that is the "key" field name to search for a match against the keyValue field. This "key" field name can be any of the field names defined through the IIR Database Administration utility.
queryOp	Parameter of string type that is the comparison operator for the query. Valid query criteria are- " < " -Values of fieldName that are less than keyValue " < = " -Values of fieldName that are less than or equal to keyValue " = " -Values of fieldName that are equal to keyValue " > " -Values of fieldName that are greater than keyValue " > = " Values of fieldName that are greater than or equal to keyValue
keyValue	Parameter of string type to be

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
compared to fieldName values in
the DB

Return Value

If successful, this function returns 0. If all parameters are valid, but no record is found to match the search criteria, the function will return - 4. For an invalid database handle, this function returns an integer value of - 5008. If any of the parameters are inconsistent or are invalid, the function will return - 5004. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve are covered in the IIR trouble shooting guide.

Remarks

A database handle must be allocated before invoking this function using the CreateDBHandle function. All functions accessing the database for specific records should not be invoked until this function has been called to successfully locate a record in the database.

Example-RunQuery

```
[variable declarations]
INVALID-HNDL = 0
hDBHandle = CreateDBHandle()
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default
```

route processing

```
if (hDBHandle == INVALID-HNDL) then
```

perform default route processing

else

```
nRet = RunQuery( hDBHandle, "AccountNo", " = ",
```

```
GetIVRDigits( hTelHandle))
```

```
# If the Query returned return OK, then print
```

out the customer information

```
if ( nRet == RETOK) then
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
1stName = GetStringFieldValue(hDBHandle,
    "LastName")

firstName = GetStringFieldValue(hDBHandle,
    "FirstName")

delCount = GetNumericFieldValue(hDBHandle,
    "DelinqCount")

process caller based upon delinquent count

and log if delinquent > 3

if (delCount > 3) then

    println "Customer" + 1stName + ",

        " + firstName + " has del =
        " + delCount

endif

endif
```

```
endif
DestroyDBHandle( hDBHandle )
```

Function SetCurrentIVRSets

The SetCurrentIVRSets function specifies for which "set" of IVR collected digits the GetIVR functions (GetIVRType, GetIVRIndicator, and GetIVRDigits) apply.

Syntax

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

destVariable = SetCurrentIVRSets(hTelHandle, setIdx)	
Part	Description
destVariable	Variable that receives the integer representing the success or failure of the function call.
h TelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
setIdx	Specifies the "set" of IVR data to access (0 based index value)

Return Value

This function returns an integer value of 0 if successful; or returns the value - 4001 for invalid telephony handle. If the setIdx parameter is invalid, it returns - 4002.

Remarks

Three functions relate to the IVR set concept, RouteMore, GetNumIVRSets, and SetCurrentIVRSets. The RouteMore function allows an IIR script to perform multiple IVR digit queries with the caller without leaving the control scope of the one IIR script.

The GetNumIVRSets allows the script to obtain the number of IVR digit sets that are potentially available, based upon the first script invocation as well as all RouteMore calls for more collected digits. The RouteMore call sends control back to a call prompting VDN to obtain more digits, returning control to the IIR script through the call vectoring adjunct command. Upon return from the RouteMore function, the most recent set of IVR collected digits is available through the GetIVR functions (GetIVRType, GetIVRIndicator, and GetIVRDigits.) The script can also access any previous "set" of IVR data (within the scope of this one IIR script) by forcing the previous "set" of IVR data to focus with the SetCurrentIVRSets function.

Example-SetCurrentIVRSets

```
[variable declarations]
NO-IVRDATA = - 1; GET-PIN-VDN = "3200"; FIRST-SET = 0;
```

```
BOTH-SETS = 2
```

```
# obtain telephony handle by converting call ID to
```

```
numeric value hTelHandle = ICRLAtoi ( $ 1 )
```

```
# check to see if account number was entered by
```

```
caller, if so collect PIN number
```

THIS PAGE BLANK (USPTO)

```
f (GetIVRType( hTelHandle ) < > NO-IVRDATA ) then
```

```
    SetPriorityCall( hTelHandle, );
```

```
    SetRouteSelected( hTelHandle, GET-PIN-VDN)
```

```
    Handle, 0);
```

```
    RouteMore( hTelHandle )
```

```
    if (GetNumIVRSets( hTelHandle ) == BOTH-SETS)
```

```
        then
```

```
        pinNum = GetIVRDigits( hTelHandle )
```

```
        SetCurrentIVRSets( hTelHandle, FIRST-SET)
```

```
        acctNum = GetIVRDigits( hTelHandle )
```

```
        Perform database match on acctNum/pinNum
```

```
    and process call
```

```
endif
```

```
endif
```

Function SetDestRoute

The SetDestRoute function is used to specify optional TAC/ARS/AAR route control information for off-PBX destinations, if the information is not included in the routeSelected. This parameter is placed in the Route Select Service message (Version 2), destRoute field.

Syntax

```
destVariable = SetDestRoute(hTelHandle, routeinfo)
```

Part	Description
destVariable	Variable that receives the integer success/failure of the call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
routeInfo	This parameter is a string containing the destRoute information.

THIS PAGE BLANK (USPTO)

Return Value

If the set of the destRoute field in the RouteSelectRequest succeeds, this function returns 0. For an invalid telephony handle, the function returns -001. If the routeInfo parameter is invalid, this function returns - 4002.

Remarks

The destRoute field of the RouteSelectRequest defines route control information for TAC/AAR/ARS. Common values for this field are "8" for AAR and "9" for ARS, however these are definable on the PBX.

Example-SetDestRoute

[variable declarations]

! obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)

! if the VDN is equal to accounting DNIS, route to

external number, use ARS

if (GetVDN(hTelHandle) == "2222") then

SetDestRoute(hTelHandle, "9")

SetRouteSelected(hTelHandle, "6125183000"

else # if VDN is marketing DNIS, route to mktg hunt

group extension

if (GetVDN(hTelHandle) == "3333") then

SetRouteSelected(hTelHandle, "3100")

endif

endif

define parameters required to route call

SetPriorityCall(hTelHandle, 0)

RouteFinal(hTelHandle)

THIS PAGE BLANK (USPTO)

Function SetDirectedAgentCallSplit

The SetDirectedAgentCallSplit defines one of the optional fields, directAgentCallSplit, in the RouteSelectRequest. This field should be provided if the routeSelected is directed to a specific agent rather than to any available agent (hunt group.) This field should contain the ACD agent's split/skill if a call is directed to a specific logged-in agent (i.e. the routeSelected is an agent login-ID rather than an extension.)

Syntax

destVariable = SetDirectedAgentCallSplit (hTelHandle, splitID)	
Part	Description
destVariable	Variable that receives the integer success/failure of the call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
split/D	This parameter is a string containing the split/skill ID where an agent specified in the routeSelected must be logged in.

Return Value

If the set of the directAgentCallSplit field in the RouteSelectRequest succeeds, this function returns 0. For an invalid telephony handle, the function returns - 4001. If the splitID parameter is invalid, this function returns - 4002.

Remarks

The splitID field of the RouteSelectRequest is only necessary if the routeSelected field has been directed to a logged in agent (agent login-ID), rather than an equipment or service extension.

Example-SetDirectedAgentCallSplit

```
[variable declarations]
# obtain telephony handle by converting call ID to

numeric value hTelHandle = ICRLAtoi ($ 1 )

# if the VDN is equal to accounting DNIS, route to

agent login-ID 3501
```

THIS PAGE BLANK (USPTO)

```
Pat. No. 5870464, *
if (GetVDN( hTelHandle ) == "2222" ) then

    SetRouteSelected( hTelHandle, "3501")
    # The specified agent login-ID (3501) will be

        logged into skill hunt group "2000"

    SetDirectedAgentCallSplit( hTelHandle, "2000")

else # if VDN is marketing DNIS, route to mktg hunt

    group extension
    if (GetVDN( hTelHandle ) == "3333" ) then

        SetRouteSelected( hTelHandle, "3100")

    endif

endif

# define parameters required to route call
SetPriorityCall( hTelHandle, 0 )
RouteFinal( hTelHandle )
```

Function SetNumericFieldValue

The SetNumericFieldValue function sets the specified field name of the "current" database record to the integer value passed in as a parameter. A current database record must be identified using the RunQuery operation before accessing fields within the record.

Syntax

```
destVariable = SetNumericFieldValue(hDBHandle, fieldName, setValue)
```

Part	Description
destVariable	Variable receiving the string field result from the operation
hDBHandle	Variable that receives the allocated database handle
fieldName	Parameter of string type containing record field name to retrieve
setValue	Parameter of integer type defining value to set the field to

THIS FACE BLANK (USPTO)

Return Value

Returns 0 if successful, - 5008 for an invalid database handle. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve are covered in the IIR trouble shooting guide.

Remarks

The SetNumericFieldValue function cannot be called before a database handle being obtained through the CreateDBHandle function, and a "current" record being defined through the RunQuery function call.

Example-SetNumericFieldValue

```
{variable declarations}
INVALID-HNDL = 0; RETOK = 0; END-OF-FILE = - 9
hDBHandle = CreateDBHandle()
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default
```

```
route processing
```

```
if ( hDBHandle == INVALID - HNDL) then
```

```
perform default route processing
```

```
else
```

```
nRet = RunQuery( hDBHandle, "AccountNo", " = ",
```

```
GetIVRDigits( hTelHandle
```

```
# while the field is equal to the key, set
```

```
acctFound field
```

```
do
```

```
SetNumericFieldValue( hDBHandle,
```

```
"AccCount", 2 )
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
UpdateRecord( hDBHandle )
nRet = MoveNextRecord( hDBHandle )
if (nRet == RETOK) then
```

```
    acctNo = GetStringFieldValue
```

```
        (hDBHandle, "AccountNo")
```

```
endif
```

```
while (nRet < > END-OF-FILE And acctNo ==
```

```
    GetIVRDigits( hTelHandle ) )
```

```
endif
route call to default call processing VDN
DestroyDBHandle( hDBHandle )
```

Function SetOutgoingUII

The SetOutgoingUII function allows the IIR script to associate caller information, up to 32 bytes, with a call. This information can be any meaningful data (such as account no, social security number, etc.) that is an alphanumeric string. It is propagated with the call whether the call is routed locally or is interflowed to another switch. Providing expected data to an awaiting VDN if the data is known in the IIR script.

Syntax

```
destVariable = SetOutgoingUII(hTelHandle, UIIType, UIILength, UIIData)
```

Part	Description
destVariable	Variable that receives the integer success/failure of the call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
UIIType	This parameter is an integer value that specifies the codeType. The possible values are as follows: - 1 = UII-NONE (indicates UII not specified) 0 = UII-USER-SPECIFIC (not supported for IIR release 1.0) 4 = UII-IA5-ASCII (ASCII string supported by IIR)
UIILength	This parameter is an integer

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
value that represents the string
length stored in the UUIData
field (does not count null
terminator.)
This parameter is a string of up
to 32 bytes in length containing
alphanumeric data to be the UUI
associated with this call record.

UUIData

Return Value

If the set of the UserToUserInfo field in the RouteSelectRequest succeeds, this function returns 0. For an invalid telephony handle, the function returns -4001. If any of the parameters are inconsistent or invalid, this function returns -4002.

Remarks

The UserToUserInfo structure in the Route Select structure can be used to send along with the call pertinent information concerning the call or caller. This data field is generic and is sent along with a call transfer either locally on the PBX or when interflowed. It is important to understand that this information must be understood by the destination, whether that be another IIR script or other adjunct, and the user must make sure that the destinations for the call routing are not expecting other UserToUserInfo in this field.

The initial version of the IIR only supports the UUI-IA5-ASCII (4) UUType. The IIR scripts are not able to manipulate binary data in this release.

Example-SetOutgoingUUI
[variable declarations]
UUI - ASCII = 4; COLL-VDN = "3000"
obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)
if the VDN is equal to accounting DNIS, route to

VDN that will collect phone number

if (GetVDN(hTelHandle) == "2222") then

SetRouteSelected(hTelHandle, COLL-VDN)
ANI = GetCallingDevice(hTelHandle)
if the callers ANI is known pass it to the

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

next IIR script

if (ANI < > "") then

SetOutgoingUUI(hTelHandle, UI-ASCII,

ICRLStrLen (ANI), ANI)

endif

endif
define parameters required to route call
SetPriorityCall(hTelHandle, 0)
RouteFinal(hTelHandle)

Function SetPriority Call

The SetPriorityCall function is used to specify the priority calling field in the RouteSelectRequest. This parameter is placed in the Route Select Service message (Version 2), priorityCalling, field and is a required component of the message before the IIR can send the Route Select message.

Syntax

destVariable = SetPriorityCall(hTelHandle, priority)	
Part	Description
destVariable	Variable that receives the integer success/failure of the call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
routeSelected	This parameter is an integer specifying the priority of the on-PBX call as either "On" (1) or "Off" (0).

Return Value

If the set of the priorityCalling field in the RouteSelectRequest succeeds, this function returns 0. For an invalid telephony handle, the function returns - 4001. If the routeSelected parameter is invalid, this function returns - 4002.

Remarks

THIS PAGE BLANK (USPTO)

The priorityCalling field is one of two mandatory fields that must be set before the route request being returned from the IIR. This function must be called before calling either of the RouteFinal or RouteMore functions.

The value must be set to "Off" (0) for all off-PBX specified destinations in the routeSelected field. If this is not done, the call will be denied.

Example-SetPriorityCall

[variable declarations]

obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)

if the VDN is equal to accounting DNIS, route to

acct hunt group extension

if (GetVDN(hTelHandle) == "2222") then

SetRouteSelected(hTelHandle, "3000")

else # if VDN is marketing DNIS, route to mktg hunt

group extension

if (GetVDN(hTelHandle) == "3333") then

SetRouteSelected(hTelHandle, "3100")

endif

endif

define parameters required to route call

SetPriorityCall(hTelHandle, 0)

RouteFinal(hTelHandle)

Function SetRoute Selected

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

The SetRouteSelected function is used to specify the destination route field in the RouteSelectRequest, which can be an extension, VDN, hunt group, or external number. This parameter is placed in the Route Select Service message (Version 2), routeSelected field and is a required component of the message before the IIR can send the Route Select message.

Syntax

destVariable = SetRouteSelected(hTelHandle, routeSelected)	
Part	Description
destVariable	Variable that receives the integer success/failure of the call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
routeSelected	This parameter is a string containing the routing specification for the call. This can be an extension, or if an off-PBX number, it can also contain the TAC/ARS/AAR route control information.

Return Value

If the set of the routeSelected field in the RouteSelectRequest succeeds, this function returns 0. For an invalid telephony handle, the function returns - 4001. If the routeSelected parameter is invalid, this function returns - 4002.

Remarks

The routeSelected field of the RouteSelectRequest is one of two mandatory fields that must be set before the route request being returned from the IIR. This function must be called before calling either of the RouteFinal or RouteMore functions. The data provided on this field can be an extension (representing an extension, agent login-ID, hunt group, or VDN), or if an off-PBX number, can also contain the TAC/ARS/AAR route control information.

Example-SetRouteSelected

```
[variable declarations]
# obtain telephony handle by converting call ID to
```

```
numeric value
```

```
hTelHandle = ICRLAtoi ( $ 1 )
# if the VDN is equal to accounting DNIS, route to
```

```
acct hunt group extension
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *
if (GetVDN(hTelHandle) == "2222") then

SetRouteSelected(hTelHandle, "3000")

else # if VDN is marketing DNIS, route to mktg hunt

group extension
if (GetVDN(hTelHandle) == "3333") then
SetRouteSelected(hTelHandle, "3100")

endif

endif

define parameters required to route call
SetPriorityCall(hTelHandle, 0)
RouteFinal(hTelHandle)

Function SetStringFieldValue

The SetStringFieldValue function sets the specified field name of the "current" database record to the string value passed in as a parameter. A current database record must be identified using the RunQuery operation before accessing fields within the record.

Syntax

Part	Description
destVariable	Variable receiving the string field result from the operation
hDBHandle	Variable that receives the allocated database handle
fieldName	Parameter of string type containing record field name to retrieve
setValue	Parameter of string type defining value to set the field to

Return Value

Returns 0 if successful, - 5008 for an invalid database handle. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve. Some of the more common errors returned by Btrieve

THIS PAGE BLANK (USPTO)

are covered in the IIR trouble shooting guide.

Remarks

The SetStringFieldValue function cannot be called before a database handle being obtained through the CreateDBHandle function, and a "current" record being defined through the RunQuery function call.

Example-SetStringFieldValue

```
[variable declarations]
INVALID-HNDL = 0; RETOK = 0; END-OF-FILE = - 9
hDBHandle = CreateDBHandle()
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default

route processing

if (hDBHandle == INVALID-HNDL) then

    perform default route processing

else

    nRet = RunQuery( hDBHandle, "AccountNo", " = ",

        GetIVRDigits( hTelHandle ) )

    # while the field is equal to the key, set

        acctFound field

    do

        SetStringFieldValue ( hDBHandle,

            "AcctNotes", "ACCT LOCATED")

        UpdateRecord( hDBHandle )
        nRet = MoveNextRecord( hDBHandle )
        if (nRet == RETOK) then

            acctNo = GetStringFieldValue
```

THIS PAGE BLANK (USPTO)

```
(hDBHandle, "AccountNo")
```

```
endif
```

```
while (nRet < > END-OF-FILE And acctNo ==
```

```
GetIVRDigits( hTelHandle ) )
```

```
endif
route call to default call processing VDN
DestroyDBHandle( hDBHandle )
```

Function SetUserProvidedCode

The SetUserProvidedCode function allows the IIR script to send code/digits (ASCII string with 0-9, *, and # only) with the routed call. These code/digits are treated as dial-ahead digits stored in the dial-ahead digit buffer. They can then be collected by subsequent collect digit vector commands on the switch. This function provides expected data to an awaiting VDN if the data is known in the IIR script.

Syntax

```
destVariable = SetUserProvidedCode( hTelHandle, codeType, codeData)
```

Part	Description
destVariable	Variable that receives the integer success/failure of the call.
hTelHandle	Telephony handle obtained from ICRLAtoi (\$ 1) function call.
codeType	This parameter is an integer value that specifies the codeType. Values include: 0 = UP-NONE (indicates UPC not present) 17 = UP-DATA-BASE-PROVIDED (IIR input, for instance)
codeData	String of up to 24 numeric characters making up code/digits to be placed in dial-ahead digit buffer upon RouteSelect (0-9, *, #)

THIS PAGE BLANK (USPTO)

Return Value

If the set of the UserProvidedCode field in the RouteSelectRequest succeeds, this function returns 0. For an invalid telephony handle, the function returns -4001. If any of the parameters are inconsistent or invalid, this function returns -4002.

Remarks

The UserProvidedCode structure in the Route Select structure can be used to "feed" a subsequent VDN/vector known code/digits that would otherwise be collected from the caller. In the case where the data is known by the IIR, there is no need to prompt the caller for the data, and with this method the caller would not be prompted in the subsequent vector if the digits were already placed there for the caller.

Example-SetUserProvidedCode

[variable declarations]

DB-PROVIDED = 17; COLL-VDN = "3000"

obtain telephony handle by converting call ID to

numeric value

hTelHandle = ICRLAtoi (\$ 1)

if the VDN is equal to accounting DNIS, route to

VDN that will collect phone number

if (GetVDN(hTelHandle) == "2222") then

SetRouteSelected(hTelHandle, COLL-VDN)

ANI = GetCallingDevice(hTelHandle)

if the callers ANI is known, go ahead and put

in type ahead buffer

if (ANI < > "") then

SetUserProvidedCode(hTelHandle,

DB-PROVIDED, ANI)

endif

THIS PAGE BLANK (USPTO)

```
endif
: define parameters required to route call
setPriorityCall( hTelHandle, 0 )
routeFinal( hTelHandle )
```

Function UpdateRecord

The UpdateRecord function updates a record in the database that has been located through the RunQuery or MoveNextRecord/MovePreviousRecord functions. The record to be updated, once located, will have the targeted fields modified through the SetStringFieldValue and SetNumericFieldValue functions, before update.

Syntax

```
destVariable = UpdateRecord(hDBHandle)
Part
destVariable      Description
                  Variable receiving integer value
                  representing success/failure of
                  call.
hDBHandle         Variable that receives the
                  allocated database handle
```

Return Value

Returns 0 if successful, for an invalid database handle the function returns - 5008. This function can also return errors in the range of - 1 to - 2000, which are negated error codes returned by Btrieve.

Remarks

The UpdateRecord function cannot be called before obtaining a database handle through the CreateDBHandle function, and defining a "current" record through the RunQuery function call.

The fields that are intended for update can be modified using the SetStringFieldValue or SetNumericFieldValue. Once the fields have been modified, the record is updated in the database by calling this function, UpdateRecord.

Example-UpdateRecord

```
[variable declarations]
INVALID-HNDL = 0; RETOK = 0
hDBHandle = CreateDBHandle
hTelHandle = ICRLAtoi ($ 1)
# If unable to create a database handle, do default
```

route processing

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

```
if (hDBHandle == INVALID - HNDL) then

    perform default route processing

else

    nRet = RunQuery( hDBHandle, "AccountNo", " = ",

        GetIVRDigits( hTelHandle ) )

    # If the Query returned return OK, then modify

        the record and update it

    if ( nRet == RETOK ) then

        SetStringFieldValue (hDBHandle,

            "CustStatus", "LOCATED")

        if (UpdateRecord(hDBHandle) < > RETOK) then

            process error on updating record

        endif

    endif

endif

DestroyDBHandle( hDBHandle )
```

TABLE 17

ACD	(Automatic Call Distribution) Phone system designed to answer incoming calls and distribute the calls based on instructions in the database.
Administrator	Designated person who is responsible for defining Call Treatment Tables and other program setup features.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

Person who is a member of an ACD split extension.
Physical device, usually a telephone or a headset.

Agent
Agent
Instrument
Agent Status

Current status of an Agent (Logged In/Out, Work Ready/Not Ready, etc.).

ANI

(Automatic Number Identification) The information passed to the switch, identifying incoming calls. If you also have Caller ID, you can then view the ANI.

Busy Hour

Hour of the day during which a telephone system carries the most traffic.

Call Treatment
Call Vector

see Call Vector.
Method that manages inbound calls via the use of routing tables.

Caller ID

Service provided by the local and long distance phone companies that identifies and displays information (name and phone number) about an incoming call. See also ANI.

CDR

(Call Detail Record) Telephone system feature which collects and records information on phone calls, such as phone number, duration, time, date.

CNE

(Certified NetWare Engineer) Designation awarded by Novell, Inc. to

CTT
DEFINITY
DNIS

persons who pass the Novell rating.
(Call Treatment Table [see Call Vector])
AT&T brand of telephone switching systems.
(Dialed Number Identification Service) For 800 numbers, this service identifies the number the caller dialed for routing purposes.

EAS
GUI

(Expert Agent Selection)
(Graphical User Interface) Standard Windows presentation of application information.

Hunt Group
IIR

Ordered group of stations.
(Intelligent Information Router) Telephony application that routes incoming calls based on a rules database.

IIR Client

IIR system components that run on Windows desktop computers.

IIR
Administrator

IIR-defined Administrator.

IIR Agent
IIR-Route

IIR-defined call center Agents.
Call routed via the IIR engine (also see IIR Rule).

IIR Server
II Digits

IIR system components that run on network servers.
Call origination Digits/Identification

ISDN

(Integrated Services Digital Network)

IVR Digits

Interactive Voice Response information collected during call vectoring (i.e. prompted or collected digits)
(Netware Loadable Module)

NLM
Novell NetWare
PBX

Novell's network server software.
(Private Branch Exchange) Telephone system within an organization, which supports and operates all the phones and the telephony switch.

PBX-Route
Router

Call routed via the DEFINITY PBX (also see PBX vector) see IIR.

Routing Path

The flow (path) a call takes to arrive at final destination.

THIS PAGE BLANK (USPTO)

Rule Station Step	Pat. No. 5870464, * Defined set of trigger events and match criteria. Physical location of an Agent and Agent instrument. Software.
-------------------------	--

APPENDIX D-ERROR MESSAGES AND CODES

The following description of error messages and codes answers common questions about installing and using the Intelligent Information Router.

IIR Simulator Errors as Shown in Table 18

Important: Win 32s is required to run the IIR Simulator. Make sure that you have correctly installed Win 32/s before attempting to run the IIR Simulator.

TABLE 18

Error Message Runtime Error: Unknown function: TelSimulate	Resolution The Simulator did not start correctly. Stop the application and restart. You should see messages similar to the following in the output window: AutoInit in progress. ICRMSG load succeeded. ICRDB load succeeded. DBinit Succeeded. ICRTEL load succeeded. ICRTelInitialize returned 0. AutoInit Done. Resolve the error before continuing.
DBInit failed.	When the Simulator starts, the software connects to the Btrieve database referenced in the ASI.INI file. If this process fails, you will see a DBInit failed message. This message will be followed by an error Return Code. If the return code is - 1 to - 2000, it is a Btrieve error. Refer to your Btrieve manual for more information. If the any other value is returned, it is an IIR error. Common problems: Verify that the IIRS Simulator Database section in your ASI.INI file (located in your windows directory) references the correct location, and that a database exists in that location. (Btrieve error 12) If your database is on a network drive, verify that you still have access to that drive. (Btrieve error 12) Make sure that Btrieve is running on the system that owns the drive referenced as your database

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

location. For a local database, you must be running Btrieve on your workstation. (Btrieve error 20)
Your Windows resources are low. Close some of the applications that are running and restart Windows to clear this error. After you have freed some of your Windows resources, restart the IIR simulator.
Your script is missing a carriage return or a semicolon ";" after the last statement in your script. The IIR Script Language uses these delimiters to identify statements. Add a Carriage Return or ";" to the end of the last statement in your script.
This message occurs if you are passing the wrong type of parameter to a command. For example, When using the RunQuery command and you pass a 5 rather than "5", even if the field you are matching against is a numeric field, you must pass the value enclosed in double quotation marks. Enclose all string parameters for all commands in double quotation marks, including integers, such as 5. Passing a string to a command that expects an integer will also cause unpredictable results. For instance, if you are using the SetNumericFieldValue command and you pass "5" rather than 5, strange things may happen.
Check all Command parameters and make sure that you are using the correct type.
One of the variables used as an input

Ran out of memory allocating symbol pOS.

Expected Delimiter, but got End-Of-File

Some of the Script statements are not being executed; no error is returned. Strange things are happening.

RunTime error S you can use the IIR Simulator. for the Simulator is the Default DOS Application Icon.

IIR Database Administration Tool as Shown by Table 19

Important: ODBC is required to run the Database Administration tool. Make sure that you have installed ODBC correctly and that you have an entry in your ODBC setup window (from the Windows Control panel) for ICRBTR.

TABLE 19

Error Message	Resolution
Data source name not found. No default driver specified.	ODBC is required to run the Database Administration Tool. You may not have installed the ODBC drivers correctly. Check the ODBC drivers: Open the ODBC setup window from the Windows Control

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

Panel. ICRBTR is the Data Source Name.
The Database directory should point to
the database location referenced in the
IIR Simulator Database section of your
ASI.INI file found in either your
Windows directory for testing, or in the
IIR database subdirectory on the IIR
server for production setup.

Couldn't find object
(unknown).

ODBC is required to run the Database
Administration
Tool.

Open ODBC setup window from the Windows
Control Panel. ICRBTR is the Data Source
Name. The Database directory should
point to the database location
referenced in the IIR Simulator Database
section of your ASI.INI file found in
your Windows directory for testing. Or
the IIR/DbData subdirectory on the IIR
server for production setup.

1. Verify that the ODBC directory
points to a location that contains
a File.DDF file.
2. Make sure that you have access to
the drive that contains the
File.DDF file.

IIR Agent Tool as Shown by Table 20

Important: ODBC is required to run the Database Administration tool. Make
sure that you have installed ODBC correctly and that you have an entry in your
ODBC setup window (from the Windows Control panel) for ICRBTR.

TABLE 20

Error Message

Data source name not found no
default driver specified.

Resolution

ODBC is required to run the Database
Administration Tool.
Open ODBC setup window from the
Windows Control Panel. ICRBTR is the
Data Source Name. The Database
directory should point to the database
location referenced in the IIR Simulator
Database section of your ASI.INI file
found in your Windows directory for
testing, or in the IIR/DbData
subdirectory on the IIR server for
production setup.

Error

Couldn't find object (unknown).

Resolution

ODBC is required to run the Database
Administration Tool. Open ODBC setup
window from the Windows Control
Panel. ICRBTR is the Data Source
Name. The Database directory should
point to the database location referenced

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

LEXSEE

in the IIR Simulator Database section of your ASI.INI file found in your Windows directory for testing, or in the IIR/DbData subdirectory on the IIR server for production setup.

1. Verify that the ODBC directory points to a location that contains a File.DDF file.
2. Make sure that you have access to the drive that contains the File.DDF file.

Error Codes as Shown by Table 21

Time Command Errors

TABLE 21

Error Code	Description	Possible Resolution
- 6	Invalid Handle	This error is returned from the Get Time commands which return integer values when the handle passed as input to the command was not a valid time handle. A time handle is created by using the CreateTimeHandle(). The value returned from this command should be passed as an input parameter to all Get Time commands.
" < INVALID TIME > "	Invalid Time	This error is returned from the GetAscTime command, usually when the parameter passed to the command was not a valid time handle. A time handle is created by using the CreateTimeHandle(). The value returned from this command should be passed as an input parameter to all Get Time commands.

Database Error Codes as Shown by Table 22

Database commands can return two types of errors: Btrieve Errors and IIR Errors. If the error code returned has a value of - 1 to - 2000, it is a Btrieve error. Some common Btrieve errors are listed below. Refer to your Btrieve Manuals for a complete listing of errors and resolutions. The error codes are listed in the Btrieve Manuals as positive rather than negative values.

TABLE 22

Error Code	Description	Possible Resolution
- 4	The application	There are no records in the database

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

cannot find the key value.	that meet the match criterion you have specified in your RunQuery command.
*	If this error is returned while
*	running the Simulator, make sure that
*	the IIR Simulator Database section of
*	your ASI.INI file found in your
*	windows directory point to the
*	correct database.
- 5	The record has a primary key defined, and the record being inserted or updated has a key value matching a record that already exists in the database.
- 12	Btrieve cannot find the specified file. This usually means one of two things: The IIR Simulator Database section in your ASI.INI file located in your Windows directory is not correct.
*	You no longer have access to that networked drive.
*	
- 18	The disk is full. You are out of space on the Drive where the database resides. If you are running the simulator, Check the IIR Simulator Database section of your ASI.INI file located in your windows directory to determine the database location. If this error was returned from the IIR, you need to free up space on the IIR drive.
*	
*	
*	
*	
*	
*	
- 20	The Record Manager or Requester is inactive. This usually means that the database you are referencing resides on a machine that does not have Btrieve loaded. You either need to start Btrieve on that machine, or move the database to a machine where Btrieve is running.
*	
*	
*	

The following Table 23 contains a list of the IIR Database errors and possible resolutions:

TABLE 23

Error Code	Description	Possible Resolution
- 5001	Out of Memory	If this error occurs while running the script under the simulator, then your Windows resources are low.
	*	You may need to stop some of the applications that are running and restart Windows to clear this error.
	*	Once you have freed some of your Windows resources, restart the IIR simulator.
	*	If this error occurs when the script is run from the IIR, then IIR server
	*	

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

* is out of memory. It may be
* necessary to stop some of the other
* applications running on the machine
* to free up the necessary memory. If
* this problem happens repeatedly you
* may need to add more memory to the
* machine.

- 5002 Invalid File Name
* The full path to the database file
* name must be 18 characters or less.
* Move your database to a directory
* structure with a shorter name, and
* update the IIR Simulator Database
* section in your ASI.INI file located
* in your Windows directory to point
* to the new location.
* You must stop and re-start the IIR
* simulator to pick up the new
* database path.

- 5004 Invalid Where Clause
* One of the match condition
* parameters passed to the RunQuery
* command is invalid.
* Verify that all the match condition
* parameters passed to the RunQuery
* command are enclosed by double
* quotes. Even integer values must be
* enclosed by double quotes.
* Example: RunQuery (hDatabase,
* "Field16, " = ", "5")

- 5005 Invalid Relational Operand
* The relational operand used in the
* RunQuery command is not valid.
* Check to make sure that the
* RunQuery command is formatted
* property.

- 5006 Field Name not Found
* The Field Name used in the
* RunQuery command does not match
* the Field Names currently defined for
* the Database.
* Use the Database Administration tool
* to find out the currently defined Field
* Names. Make sure that the Database
* Administration tool points to the
* same database that the script is
* running against.
* Use the Windows Control
* Panel / ODBC setup to determine the
* database location for the Database
* Administration Tool (ICRBTR). If
* you are running the simulator, check
* the IIR Simulator Database section of
* the ASI.INI file found in your
* Windows directory. If you
* encountered this error while running
* from the IIR, point the Database
* Administration tool to the production
* database located in the IIR/dbdata
* subdirectory on the IIR server.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

* Notes: If you change the Database
* Field Names, you must stop and
* re-start the simulator and IIR NLM
* before the new Field Names are
* available to a script.
* Database Field Names are case
* sensitive. Make sure the Field Names
* used in the Script match exactly on the
* left.

- 5009 Not Implemented

* This command has not yet been
* implemented. Remove this command
* from your script.

- 5006 Field Name not Found

* The Field Name used in the
* RunQuery command does not match
* the Field Names currently defined for
* the Database.
* Use the Database Administration tool
* to find out the currently defined Field
* Names. Make sure that the Database
* Administration tool points to the
* same database that the script is
* running against.
* Use the Windows Control
* Panel / ODBC setup to determine the
* database location for the Database
* Administration Tool (ICRBTR). If
* you are running the simulator, check
* the IIR Simulator Database section of
* the ASI.INI file found in your
* Windows directory. If you
* encountered this error while running
* from the IIR, point the Database
* Administration tool to the production
* database located in the IIR/dbdata
* subdirectory on the IIR server.

* Notes: If you change the Database
* Field Names, you must stop and
* re-start the simulator and IIR NLM
* before the new Field Names are
* available to a script.
* Database Field Names are case
* sensitive. Make sure the Field Names
* used in the Script match exactly the
* Field Names defined in the Database
* Administration Tool

- 5010 Field Data Too Large

* The database supports a maximum
* field size of 100 characters. Make
* sure that the Value passed in the
* SetStringFieldValue is not more than
* 100 characters.

- 5011 Label Database is Corrupted

* The Labeldef.DTA file has been
* corrupted.
* If running the simulator, verify that
* the IIR Simulator Database section in
* the ASI.INI file in your windows
* directory points to the correct

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

location. If this is correct, it may be necessary to restore this file from a backup. This file contains the Field Names for the database. If you do not have a backup of this file, you will need to perform the following steps to create an empty Labeldef.DTA file.

1. Backup your production database.
2. Re-install the IIR NLM. This will create an empty database.
3. Save off the empty Labeldef.DTA file.

4. Restore the previously backup database.
5. Copy the empty Labeldef.DTA file over the corrupted file.

6. Use the Database Administration Tool to define your database labels. This error is returned from the GetStringFieldValue command.

GetStringFieldValue command:
There are two common causes for
this error:

this error:
Database Handle passed is invalid.
Check to see that your Database
Handle is valid. Use the
CreateDbHandle command to create
the database handle. If your a handle
was created by the CreateDbHandle
command, ion for the Database
Administration Tool (ICRBTR). If
you are running the simulator, check
the IIR Simulator Database section of
the ASI. INI file found in your
Windows directory. If this error was
encountered while running from the
IIR, point the Database

IIR, point the Database Administration tool to the Notes: If you change the Database Field Names, you must stop and re-start the simulator and IIR NLM before the new Field Names are available to a script.

Database Field Names are case sensitive. Make sure the Field Names used in the Script match exactly the Field Names defined in the Database Administration Tool

This error is returned from the `GetNumericFieldValue` command. There are two common causes for this error:

this error:
Database Handle passed is invalid.
Check to make sure that the Database
Handle you are passing is a valid
handle. Use the CreateDbHandle

```
GetStringField
Value failed
```

```
GetNumericFieldV
alue failed
```

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

* command to create the database
* handle. If you are using a handle that
* was created by the CreateDbHandle
* command, make sure that your script
* is not altering the value of the
* variable. Make sure you are using
* " = = " (double =) for all comparison
* operations. A single = will change
* the value of the variable on the left.
* The Field Name passed is not a valid
* field name in the database. Use the
* Database Administration tool to find
* out the currently defined Field
* Names. Make sure that the Database
* Administration tool is pointed to the
* same database that the script is
* running against. Use the Windows
* Control Panel ODBC setup to
* determine the database location for
* the Database Administration Tool
* (ICRBTR). If you are running the
* simulator, check the IIR Simulator
* Database section of the ASI.INI file
* found in your Windows directory. If
* this error was encountered while
* running from the IIR, point the
* Database Administration tool to the
* production database located in the
* IIR/DBDATA sub-directory on the
* IIR server.
* Notes:
* If you change the Database Field
* Names, you must stop and re-start
* the simulator and IIR
*

Telephony Error Codes

Telephony commands can return four types of errors: Local ACS Errors, Tserver ACS Errors, Tserver CSTA Errors, and IIR Errors. Note: Any object used as the Destination for a Route command is considered a device. The device may be a VDN, Agent Extension, Agent Login ID, or Hunt Group. All Devices must be administered using the Telephony Services Administration tool before they are recognized by the IIR.

VDNs used to accept Inbound calls and VDNs used as the destination of a RouteMore command must have Route Sessions initiated before being recognized by the IIR. Use the VDN Administration Tool to start and stop Route Sessions.

Local ACS Errors

If the error code returned has a value of - 1 to - 99, it is a Local ACS error. Refer to your Telephony Services Manuals for a description and resolution.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

Server ACS Errors

If the error code returned has a value of 100 to 1999, it is a Tserver ACS error. Refer to your Telephony Services Manuals for a description and resolution. The errors are listed in the Telephony Services manual under an Error Code that is 100 less than the error code returned by the IIR. In Example 30, if 100 is returned by the IIR, look for 0 in the Telephony Services Manuals.

Example 29 Error Code	Description	Possible Resolution
	TSAPI (27) No Device Record Found.	This error usually means that the Device (VDN, Agent Extension, Agent Login, Hunt Group) has not been setup using the Telephony Service Administration Tool.
*	*	All devices must be administered using the Telephony Services Administration tool before they are recognized by the IIR. This includes Agent Login Ids, Agent Extensions, VDNs and HuntX Groups.
*	*	
*	*	
*	*	
*	*	
*	*	
*	*	

CSTA Tserver Errors

If the error code returned has a value of 2000 to 2999, it is a Tserver CSTA error. Some common CSTA errors are listed below. Refer to your Telephony Services manuals for a complete listing of errors and resolutions. The errors are listed in the Telephony Services manual under an Error Code that is 2000 less than the error code returned by the IIR. In Example 30, if 2000 is returned by the IIR, look for 0 in the Telephony Services Manuals.

Common CSTA Errors

EXAMPLE 30 Error Code	Description	Possible Resolution
2012	TSAPI (12) An invalid device identifier has been specified in the device parameter.	This error can be returned from any Telephony Command which requires a device as an input parameter. For example, QueryAgentState requires a valid agent extension as an input parameter. If the input parameter is not a valid agent extension, this error will be returned.
*	*	All devices must be administered using the Telephony Services Administration tool before they are recognized by the IIR. This includes Agent Login Ids, Agent Extensions, VDNs and Hunt Groups.
*	*	This error can be returned when the IIR
*	*	
*	*	
*	*	
*	*	
*	*	
2044	(TSAPI 44)	

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

The specified routing device already has a registered routing server.
*
*

is trying to initiate a routing session for a VDN. If there is already a routing session in process for that VDN, the a request will fail. You must stop the session that is already in process before you can initiate a new session for that VDN.

IIR Telephony Errors

The following Table 24 contains a list of the IIR Telephony errors and possible resolutions:

Error Code	Description	Possible Resolution
4000	Generic Failure * * *	An unexpected error has occurred. It may be necessary to stop and restart the IIR. If this problem happens repeatedly contact Technical Support.
4001	Invalid Telephony Handle * * * * * * * * *	The telephony handle passed in the Command is invalid. Use the TELAtoi command to convert the \$ 1 parameter to the telephony handle to be used in Telephony commands. If you are using a handle that was created by the TELAtoi command, make sure that your script is not altering the value of the variable. Make sure you are using " = = " (double =) for all comparison operations. A single will change the value of the variable on the left.
4002	Invalid parameter * *	One of the parameters passed to the command is invalid. Make sure that all of the parameters are of the appropriate type, Strings, Integers or Variables.
4003	Wait Structure Allocate Failure * * *	The IIR server is out of memory. It may be necessary to stop some of the other applications running on the machine to free up the necessary memory. If this problem happens repeatedly you may need to add more memory to the machine.
4004	Event Allocate Failure * * * *	The IIR server is out of memory. It may be necessary to stop some of the other applications running on the machine to free up the necessary memory. If this problem happens repeatedly you may need to add more memory to the machine.
4005	Ini file failure * * * * *	IIR is unable to read/write necessary information to the CTI.CFG file. Verify that the file exists on the IIR server and that the information in the file is correct. If the problem persists it may be necessary to restore this file from a Backup, or re-install the IIR NLM.

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

4006	Begin Thread Failure * * * *	The IIR server is out of memory. It may be necessary to stop some of the other applications running on the machine to free up the necessary memory. If this problem happens repeatedly you may need to add more memory to the machine.
4007	Wait on Event Time out * * * *	The command did not receive the expected response event within 10 seconds. This normally means there is some type of problem with the Tserver. It may be necessary to stop and start the Tserver and IIR server to correct the problem.
4008	No Valid ACS connection * *	The IIR has lost it's telephony connection. It automatically tries to reconnect based on the settings in the CTI.CFG file.
4009	Memory Allocation Failed * * * * * * * * * * * *	The IIR server is out of memory. It may be necessary to stop some of the other applications running on the machine to free up the necessary memory. If this problem happens repeatedly you may need to add more memory to the can be associated with only one script. Make sure that the VDN you are trying to define is not already defined. If you do not see an entry for the VDN, use the refresh option to re-sync the display with the IIR setup. If you still have a problem, you need to stop and re-start the VDN Administration tool. If the problem persists it may be necessary to stop and re-start the IIR.
- 4012	Create Script Failed * * * * * * * * * * *	This is an IIR runtime error. It is not returned in response to a command. If this error appears in the IIR log file, use the VDN administration tool to verify that the correct scripts are "turned on", and verify that these scripts exists in the IIR directory on the IIR server. If the VDN configuration is correct and the Scripts are located in the appropriate directory, try stopping and re-starting the session(s) for the script referenced in the Error Message. If this fails, it may be necessary to stop and re-start the IIR.
4013	Execute Script Failed * * * * * * * *	This is an IIR runtime error. It is not returned in response to a command. If this error appears in the IIR log file, use the VDN administration tool to verify that the correct scripts are "turned on". If the setup is correct, you may have a logic problem in your script. Use the Simulator to verify that your script logic is correct. If the script appears to be correct, try stopping and re-

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

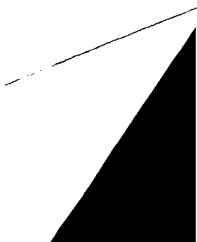
* starting the VDN session(s) for the
* script referenced in the Error Message.
* If this fails, it may be necessary to
* stop and re-start the IIR.

Miscellaneous Low Level Error Codes are Shown by Table 25

TABLE 25

Error Code	Description	Possible Resolution
- 200	Unable to Open Log File	If you are running the Simulator, check the Log section of your ASI.INI file located in the windows directory to determine your Log File location. Make sure that you are Read/Write access to this Drive/File.
		If this error is returned as a Run Time error from the IIR, check Log section in the CTI.CFG file located on the IIR server to determine the log file location. Make sure that the IIR server has Read/Write access to this Drive/File.
- 201	Disk Full	If you are running the Simulator, check the Log section of ASI.INI file located in the windows directory to determine your Log File location. Make sure that disk space is available on this drive. If this error is returned as a Run Time error from the IIR, check Log section in the CTI.CFG file located on the IIR server to determine the log file location. Make sure that disk space is available on this drive.
- 202	Invalid Resource Name	Low level error. This error is usually accompanied by other errors in the log file which provide more specific information.
- 203	Invalid Resource Size	Low level error. This error is usually accompanied by other errors in the log file which provide more specific information.
- 204	Insufficient Memory	If this error occurs while running the simulator, then your Windows resources are low. You may need to stop some of the applications that are running and restart Windows to clear this error. Once you have freed some of your Windows resources, restart the IIR simulator.
		If this error occurs in the IIR,

THIS PAGE BLANK (USPTO)



Pat. No. 5870464, *

* then IIR server is out of memory.
* It may be necessary to stop some of
* the other applications running on
* the machine to free up the necessary
* memory. If this problem happens
* repeatedly you may need to add more
* memory to the machine.

- 205 Invalid Handle Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 206 Resource Not Found Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 207 Resource Already Exists Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 208 Semaphore Time-out The RouteMore command uses
Semaphores to control the amount of
the IIR waits before terminating the
command. A Semaphore time out in
response to a RouteMore command
means that the call did not complete
the VDN processing and return to the
IIR within the amount of time

\$ errors

- * in the log file which provide more
* specific information.
- 210 Semaphore not owned Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 211 Invalid Resource Sign Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 212 Resource Not Owned Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 213 Resource is Owned Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 214 Invalid Parameter Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 215 Read Time-out Low level error. This error is
usually accompanied by other errors
in the log file which provide more
specific information.
- 216 Write Time-Out Low level error. This error is
usually accompanied by other errors

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

- * in the log file which provide more
- * specific information.
- 217 Message Informational Message
- 218 Truncated
- Message File
- Missing
- * If this error is reported from the
- * IIR Simulator, check the MsgFile
- * keyword in the Log section of your
- * ASI.INI file located in the windows
- * directory. Make sure that this
- * entry points the correct file. This
- * is normally setup to point the
- * CTIERR.MSG file located in the same
- * directory as the IIR Simulator
- * software.
- * If this error is reported from the
- * IIR NLM, check the MsgFile keyword
- * in the Log section of the CTI.CFG
- * file located in the IIR directory on
- * the server. Make sure that this
- * entry points the correct file. This
- * is normally setup to point the
- * CTIERR.MSG file located in the same
- * directory as the IIR NLM software.
- 219 Check the CTIERR.MSG file and make
- Message Read Error
- sure it has not been corrupted. It
- * may be necessary to restore this
- * file from a backup, or Re-Install
- * the IIR software.
- * If this error is reported from the
- * IIR Simulator, check the MsgFile
- * keyword in the Log section of your
- * ASI.INI file located in the windows
- * directory. Make sure that this
- * entry points the correct file. This
- * is normally setup to point the
- * CTIERR.MSG file located in the same
- * directory as the IIR Simulator
- * software.
- * If this error is reported from the
- * IIR NLM, check the MsgFile keyword
- * in the Log section of the CTI.CFG
- * file located in the IIR directory on
- * the server. Make sure that this
- * entry points the correct file. This
- * is normally setup to point the
- * CTIERR.MSG file located in the same
- * di on.
- 300 Invalid Index
- * Low level error. This error is
- * usually accompanied by other errors
- * in the log file which provide more
- * specific information.
- 301 Shared Blocks
- Exceeded
- * Low level error. This error is
- * usually accompanied by other errors
- * in the log file which provide more
- * specific information.
- 302 Too Many
- Low level error. This error is

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

	Keys	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 303	Invalid Key	Low level error. This error is
	Length	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 304	Record Too	Low level error. This error is
	Small	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 305	Record Too	Low level error. This error is
	Large	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 306	Allocate	possible low memory situation.
	Record	Low level error. This error is
	Failed	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 307	Invalid Record	Low level error. This error is
	Id	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 308	Internal Error	possible low memory situation.
	in Memory	Low level error. This error is
	Database	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 309	Maximum	Low level error. This error is
	number	usually accompanied by other errors
	of records	in the log file which provide more
	exceeded	specific information.
- 310	Index already	Low level error. This error is
	exists	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 311	Duplicate Key	Low level error. This error is
	*	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 312	Record Not	Low level error. This error is
	Found	usually accompanied by other errors
	*	in the log file which provide more
	*	specific information.
- 313	Memory	Low level error. This error is
	Database	usually accompanied by other errors
	not locked	in the log file which provide more
	*	specific information.
- 314	Read only	Low level error. This error is
	memory	usually accompanied by other errors
	database	in the log file which provide more
	*	specific information.
- 315	Failed to meet	Low level error. This error is
	index criterion	usually accompanied by other errors
	*	in the log file which provide more

THIS PAGE BLANK (USPTO)

Pat. No. 5870464, *

* 316 Invalid record buffer * * 317 Not supported for Shared * * 318 Owner no longer exists * *	specific information. Low level error. This error is usually accompanied by other errors in the log file which provide more specific information. Low level error. This error is usually accompanied by other errors in the log file which provide more specific information. Low level error. This error is usually accompanied by other errors in the log file which provide more specific information.
---	---

CLAIMS: What is claimed is:

[*1] 1. An apparatus for routing an external event, comprising:

a telephony controller in communication with a telephone switching station, the telephony controller, in response to an external event of the telephone switching station, operable to create or retrieve a handle for the external event, the handle comprising a script identifier;

an external data manager in communication with an external link, the external data manager, in response to an external event of the external link, operable to create or retrieve a handle for the external event, the handle comprising a script identifier;

a script interpreter engine in communication with the telephony controller and the external data manager, the script interpreter engine, in response to receiving a script identifier, operable to create or retrieve a script handle associated with the script identifier, retrieve from a script storage a script associated with the script handle, and to invoke the script to render an output event for the external event; and

a handle manager in communication with the telephony controller, the script interpreter engine, and the external data manager, the handle manager operable to store handles and script handles.

[*2] 2. The apparatus of claim 1, further comprising a database system, the database system comprising:

a database controller in communication with the script interpreter engine and the handle manager, the database controller, in response to the output event, operable to access a database engine; and

the database engine operable to access a database storage to retrieve database information associated with the external event.

[*3] 3. The apparatus of claim 2, further comprising an agent station, the agent station comprising:

THIS PAGE BLANK (USPTO)

a communication link operable to receive the external event; and

a display operable to display the database information associated with the external event.

[*4] 4. The apparatus of claim 1, further comprising an administration station, the administration station comprising:

a router administrator operable to receive a simulator script;

a system simulator comprising;

a user interface operable to receive a simulator external event;

a simulator telephony controller in communication with the user interface, the simulator telephony controller, in response to an simulator external event for the user interface, operable to get a simulator handle associated with the simulator external event, the simulator handle comprising a simulator script identifier;

a simulator external data manager in communication with a simulator external link, the simulator external data manager, in response to a simulator external event from the simulator external link, operable to get a simulator handle associated with the simulator external event, the simulator handle comprising a simulator script identifier;

a simulator script interpreter engine in communication with the simulator telephony controller and the simulator external data manager, the simulator script interpreter engine, in response to receiving a simulator script identifier, operable to get a simulator script handle associated with the simulator script identifier, retrieve from a simulator script storage a simulator script associated with the simulator script handle, and to invoke the simulator script to render a simulator output event for said simulator external event;

a simulator handle manager in communication with the simulator telephony controller, the simulator script interpreter engine, and the simulator external data manager, the simulator handle manager operable to store simulator handles and simulator script handles; and

the router administrator operable to download the simulator script as a script to the script storage.

[*5] 5. The apparatus of claim 4, further comprising:

a database system, the database system comprising:

a database controller in communication with the script interpreter engine and the handle manager, the database controller, in response to the output event, operable to access a database engine;

the database engine operable to access a database storage to retrieve database information associated with the external event; and

THIS PAGE BLANK (USPTO)

the administration station further comprising a database administrator operable to receive information and to download the information to the database storage.

[*6] 6. The apparatus of claim 5, further comprising an agent station, the agent station comprising:

a communication link operable to receive the external event; and

a display operable to display the database information associated with the external event.

[*7] 7. The apparatus of claim 6, further comprising:

a network comprising:

a network interface in communication with the script interpretation engine and the handle manager; and

a link coupled to the agent station, the administration station, and the network interface, the link operable to communicate between the agent station, the administration station, and the network interface.

[*8] 8. The apparatus of claim 1, further comprising:

a time utility operable to commence and monitor time-based events;

a string parse utility operable to preform string searches and to parse data; and

an input/output utility operable to access the script storage.

[*9] 9. The apparatus of claim 1, the external event further comprising a telephony communication.

[*10] 10. The apparatus of claim 1, the output event further comprising a telephony route select.

[*11] 11. The apparatus of claim 1, the external event further comprising a stock price.

[*12] 12. The apparatus of claim 1, the output event further comprising a buy/sell command.

[*13] 13. The apparatus of claim 1, the script identifier further comprising a script file name.

[*14] 14. The apparatus of claim 1, further comprising:

the script handle comprising a reference count indicating a status of the script; and

the handle manager operable to delete from the script storage a script having a script handle with a predefined reference count.

THIS PAGE BLANK (USPTO)

[*15] 15. The apparatus of claim 1, further comprising:

the script handle comprising a reference count;

the script interpreter engine operable to adjust the reference count of the script handle to indicate a status of the script; and

the handle manager operable to delete from the script storage a script having a script handle with a predefined reference count.

[*16] 16. The apparatus of claim 1, further comprising:

the script handle comprising a reference count indicating a status of the script;

the script interpreter engine, in response to receiving a script identifier, operable to increment the reference count of the script handle associated with the script identifier;

the script interpreter engine, in response to invoking the script, operable to decrement the reference count of the script handle associated with the script; and

the handle manager operable to delete from the script storage a script having a script handle with a predefined reference count.

[*17] 17. An apparatus for routing a telephone communication, comprising:

a telephony controller in communication with a telephone switching station, the telephony controller, in response to a route request of the telephone switching station, operable to create or retrieve a handle for the route request, the handle comprising a script identifier;

a script interpreter engine in communication with the telephony controller, the script interpreter engine, in response to receiving a script identifier, operable to create or retrieve a script handle for the script identifier, retrieve from a script storage a script associated with the script handle, and to invoke the script to render an output event for the telephony communication; and

a handle manager in communication with the telephony controller and the script interpreter engine, the handle manager operable to store handles and script handles.

[*18] 18. The apparatus of claim 17, further comprising a database system, the database system comprising:

a database controller in communication with the script interpreter engine and the handle manager, the database controller, in response to the output event, operable to access a database engine; and

the database engine operable to access a database storage to retrieve database information associated with the telephony communication.

THIS PAGE BLANK (USPTO)

[*19] 19. The apparatus of claim 18, further comprising an agent station, the agent station comprising:

a communication link operable to receive the telephony communication; and

a display operable to display the database information associated with the telephony communication.

[*20] 20. The apparatus of claim 17, further comprising an administration station, the administration station comprising:

a router administrator operable to receive a simulator script;

a system simulator comprising;

a user interface operable to receive a simulator telephony communication;

a simulator telephony controller in communication with the user interface, the simulator telephony controller, in response to an simulator telephony communication from the user interface, operable to get a simulator handle associated with the simulator telephony communication, the simulator handle comprising a simulator script identifier;

a simulator script interpreter engine in communication with the simulator telephony controller and the simulator script interpreter engine, in response to receiving a simulator script identifier, operable to get a simulator script handle associated with the simulator script identifier, retrieve from a simulator script storage a simulator script associated with the simulator script handle, and to invoke the simulator script to render a simulator output event for said simulator telephony communication;

a simulator handle manager in communication with the simulator telephony controller and the simulator script interpreter engine, the simulator handle manager operable to store simulator handles and simulator script handles; and

the router administrator operable to download the simulator script as a script to the script storage.

[*21] 21. The apparatus of claim 20, further comprising:

a database system, the database system comprising:

a database controller in communication with the script interpreter engine and the handle manager, the database controller, in response to the output event, operable to access a database engine;

the database engine operable to access a database storage to retrieve database information associated with the telephony communication; and

the administration station further comprising a database administrator operable to receive information and to download the information to the database storage.

[*22] 22. The apparatus of claim 21, further comprising an agent station, the agent station comprising:

THIS PAGE BLANK (USPTO)

a communication link operable to receive the telephony communication; and

a display operable to display the database information associated with the telephony communication.

[*23] 23. The apparatus of claim 22, further comprising:

a network comprising:

a network interface in communication with the script interpretation engine and the handle manager; and

a link coupled to the agent station, the administration station, and the network interface, the link operable to communicate between the agent station, the administration station, and the network interface.

[*24] 24. The apparatus of claim 17, further comprising:

a time utility operable to commence and monitor time-based events;

a string parse utility operable to preform string searches and to parse data; and

an input/output utility operable to access the script storage.

[*25] 25. The apparatus of claim 17, the output event further comprising a telephony route select.

[*26] 26. The apparatus of claim 17, the script identifier further comprising a script file name.

[*27] 27. The apparatus of claim 17, further comprising:

the script handle comprising a reference count indicating a status of the script; and

the handle manager operable to delete from the script storage a script having a script handle with a predefined reference count.

[*28] 28. The apparatus of claim 17, further comprising:

the script handle comprising a reference count;

the script interpreter engine operable to adjust the reference count of the script handle to indicate a status of the script; and

the handle manager operable to delete from the script storage a script having a script handle with a predefined reference count.

[*29] 29. The apparatus of claim 17, further comprising:

the script handle comprising a reference count indicating a status of the script;

THIS PAGE BLANK (USPTO)

the script interpreter engine, in response to receiving script identifier, operable to increment the reference count of the script handle associated with the script identifier;

the script interpreter engine, in response to invoking the script, operable to decrement the reference count of the script handle associated with the script; and

the handle manager operable to delete from the script storage a script having a script handle with a predefined reference count.

[*30] 30. A method of routing a telephone communication, comprising the steps of:

receiving a route request;

creating or retrieving a telephony handle for the route request;

using the telephony handle, determining a script identifier for the route request; and

invoking a script associated with the script identifier to render an output event for the telephone communication.

[*31] 31. The method of claim 30, further comprising the steps of:

creating or retrieving a script handle for the script identifier; and

using the script handle to invoke the script.

[*32] 32. The method of claim 30, wherein the output event comprises a telephony route select.

[*33] 33. The method of claim 30, wherein the output event comprises a request for database records.

[*34] 34. The method of claim 30, wherein the script identifier comprises a script file name.

[*35] 35. A method of routing an external event, comprising the steps of:

receiving an external event;

creating or retrieving a handle for the external event;

using the handle, determining a script identifier for the external event; and

invoking a script associated with the script identifier to render an output event for the external event.

[*36] 36. The method of claim 35, further comprising the steps of:

creating or retrieving a script handle for the script identifier; and

THIS PAGE BLANK (USPTO)

using the script handle to invoke the script.

[*37] 37. The method of claim 35, wherein the external event comprises a telephony communication.

[*38] 38. The method of claim 35, wherein the output event comprises a telephony route select.

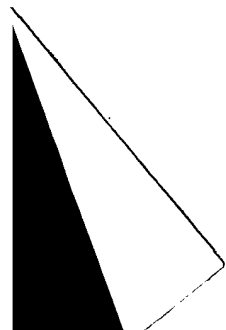
[*39] 39. The method of claim 35, wherein the external event comprises a stock price.

[*40] 40. The method of claim 35, wherein the output event comprises a buy/sell command.

[*41] 41. The method of claim 35, wherein the output event comprises a request for database records.

[*42] 42. The method of claim 35, wherein the script identifier comprises a script file name.

THIS PAGE BLANK (USPTO)



1ST PATENT of Level 1 printed in FULL format.

6,011,844

<=2> GET 1st DRAWING SHEET OF 7

Jan. 4, 2000

Point-of-presence call center management system

INVENTOR: Uppaluru, Prem, Cupertino, California
Sundaram, Mukesh, San Jose, California

ASSIGNEE-AT-ISSUE: Callnet Communications, Cambell, California (02)

APPL-N0: 249,395

FILED: Feb. 12, 1999

INT-CL: [6] H04M 3#42; H04M 7#00; H04M 3#00

US-CL: 379#220; 379#266; 379#219; 379#265; 379#221; 379#211

CL: 379

SEARCH-FLD: 379#265, 266, 219, 220, 221, 211, 309, 201, 230, 222

REF-CITED:

U.S. PATENT DOCUMENTS		
4,048,452	9/1977	* Oehring et al.
4,109,113	8/1978	* Allison, Jr. et al.
4,313,035	1/1982	* Jordan et al.
4,313,036	1/1982	* Jabara et al.
4,400,587	8/1983	* Taylor et al.
4,451,705	5/1984	* Burke et al.
4,510,351	4/1985	* Costello et al.
4,737,983	4/1988	* Frauenthal et al.
4,757,267	7/1988	* Riskin
4,788,715	11/1988	* Lee
4,847,890	7/1989	* Solomon et al.
4,878,239	10/1989	* Solomon et al.
4,893,301	1/1990	* Andrews et al.
4,924,491	5/1990	* Compton et al.
4,953,204	8/1990	* Cuschleg, Jr. et al.
4,975,945	12/1990	* Carbullido
5,020,095	5/1991	* Morganstein
5,073,890	12/1991	* Danielsen
5,164,983	11/1992	* Brown et al.
5,168,515	12/1992	* Gechter et al.
5,181,236	1/1993	* LaVallee et al.
5,185,782	2/1993	* Srinivasan
5,206,903	4/1993	* Kohler et al.
5,271,058	12/1993	* Andrews et al.
5,278,898	1/1994	* Cambray et al.
5,291,550	3/1994	* Levy et al.
5,291,552	3/1994	* Kerrigan et al.

379#265

	Pat. No. 6011844, *	
5,299,259	3/1994 *	Otto
5,311,574	5/1994 *	Livanos
5,329,583	7/1994 *	Jurgensen et al.
5,335,268	8/1994 *	Kelly, Jr. et al.
5,384,841	1/1995 *	Adams et al.
5,392,345	2/1995 *	Otto
5,459,780	10/1995 *	Sand
5,467,391	11/1995 *	Donaghue et al.
5,506,898	4/1996 *	Costantini et al.
5,519,773	5/1996 *	Dumas et al.
5,524,147	6/1996 *	Bean
5,528,678	6/1996 *	Kaplan
5,633,924	5/1997 *	Kaish et al.
5,721,770	2/1998 *	Kohler
5,881,145	3/1999 *	Giuhat et al.

379#201

379#207

PRIM-EXMR: Wolinsky, Scott

ASST-EXMR: Huynh, David

LEGAL-REP: Blakely, Sokoloff, Taylor & Zafman LLP

CORE TERMS: gateway, network, manager, remote, message, server, proxy, toll free, local call, user, inbound, script, redirected, telephone, switch, incoming, interactive, toll-free, telephony, answering, terminate, port, bridge, directory, customer, database, caller, module, pseudo, queue

ABST:

A point-of-presence (POP) call center system capable of answering, servicing, queuing and routing of calls at local points of presence to reduce communications costs and enhance operational efficiency for toll-free inbound call centers. The POP call center system includes a set of point-of-presence call center gateways distributed at points of presence close to the point of call origination that are connected by a virtual private network to premises call center gateways at business locations where the call centers reside.

NO-OF-CLAIMS: 30

EXMPL-CLAIM: <=45> 1

NO-OF-FIGURES: 7

NO-DRWNG-PP: 7

SUM:

FIELD OF THE INVENTION

The present invention relates to the field of telecommunication, and more particularly to management of toll free telephone calls.

BACKGROUND OF THE INVENTION

FIG. 1 is a functional diagram of a premises call center connecting an end user 116 to a business call center 108 via an originating Local Public

Switched Telecommunications Network (PSTN) 106, a Long Distance Network 114 and terminating Local PSTN 106. Business call centers are typically put together by integrating multiple system components into a complete business solution to answer, service, queue and route inbound customer calls. These system components can include a Private Branch Exchange (PBX) 102, an Automatic Call Distributor (ACD) 112 and an Interactive Voice Response (IVR) System 110 in addition to customer service or help desk applications for the call center agents 104. Many call centers deploy a Computer Telephony Integration (CTI) server providing intelligent call routing. Traditionally, different vendors supplied the different system components and systems integrators pulled the components together into a solution.

FIG. 2 is a functional diagram of a network-based call center connecting an end user 116 to a business call center 108 via an originating Local PSTN 106, a Long Distance Network 114 and a terminating Local PSTN 106. Network call centers may include a Switch 122, an ACD 112 and an IVR 110 within the Long Distance Network 114 and provide call answering, servicing and queuing services. These services are built on call center solutions residing inside the network that aggregate the services across multiple business customers on the shared physical configurations. Many call center vendors have targeted this fast growing network call center market with PSTN integrated systems and solutions.

The call centers depicted in FIGS. 1 and 2 each share the disadvantage that long distance toll charges accrue while a call is on hold awaiting connection to a call center agent. Long distance toll charges also accrue while the caller is interacting with the Interactive Voice Response.

SUMMARY OF THE INVENTION

A method and system for managing a toll free long distance call to a business call center are disclosed. A toll free long distance call to a business call center is redirected to a local call center. The redirected toll free call is automatically answered in the local call center to determine whether long distance connection to the business call center is necessary. If connection to the remote call center is necessary, the redirected toll free call is bridged with a telephone connection in the business call center via a long distance network.

BRIEF DESCRIPTION OF THE DRAWING

The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawing in which like references indicate similar elements and in which:

FIG. 1 is a schematic diagram of a prior art call center configuration with PBX, ACD and IVR systems located at the business call center;

FIG. 2 is a schematic diagram of a prior art network based call center configuration with Switch, ACD and IVR systems located inside the long distance network;

FIG. 3 is a schematic diagram of a POP Call Center configuration according to an embodiment that includes Switch and POP Call Center Gateway located at

points of presence connected to Premises Call Center Gateway over a Call Center Network;

FIG. 4 is a schematic diagram of components of a POP Call Center system according to one embodiment, including a POP Call Center Gateway, a Premises Call Center Gateway and a Call Center Network of a business;

FIG. 5 is a schematic diagram of components of a POP Call Center system according to an embodiment that supports a single business with multiple call center sites connected with multiple POP Call Centers;

FIG. 6 is a schematic diagram of components of a POP Call Center system according to an embodiment that supports multiple business call centers connected to multiple POP Call Centers; and

FIG. 7 is a schematic diagram of POP Call Center System software modules and their interconnections according to one embodiment.

DETDISC:

DETAILED DESCRIPTION

Although the present invention is described below by way of various embodiments that include specific structures and methods, embodiments that include alternative structures and methods may be employed without departing from the principles of the invention described herein.

OVERVIEW OF EMBODIMENTS

In general, embodiments described below feature a global call center system capable of answering, servicing, queuing and routing of calls at local points of presence to reduce communications costs and enhance operational efficiency for toll-free inbound call centers. In at least one embodiment, the global call center system includes a set of point-of-presence call center gateways distributed at points of presence close to the point of call origination that are interconnected by a virtual private network to premises call center gateways at business locations where the call centers reside.

A point-of-presence (POP) call center gateway according to embodiments of the invention is capable of intercepting and answering inbound toll-free telecommunications calls at or near the point of call origination. The POP call center gateway is also capable of providing interactive voice response based automated service, holding and queuing the calls until operators are available to service the call, and playing music or customized announcements to the caller while the call is being held. The POP call center gateway is further capable of requesting connected premises call center gateways to originate proxy calls on its behalf, monitoring call progress and routing the locally queued calls to the premises call center just in time before the operator picks up the call.

A premises call center gateway according to embodiments of the invention is capable of receiving proxy call origination requests from connected POP call center gateways and in response, generating and presenting proxy calls to the automatic call distributor (ACD) at the premises call center. The premises call center gateway is further capable of monitoring the progress of such proxy

calls within the ACD for operator availability, communicating with the appropriate POP call center gateways for just in time call delivery to the selected operator, and bridging the calls between the POP call center gateways and the premises ACD.

Embodiments of the invention include a system and method for operating the global call center system where in a customer calls an advertised inbound toll-free number to reach a business call center. The call is intercepted at the local central office or tandem switch and routed to a local POP call center gateway using the point of call routing capability of a Service Management System/800 (SMS/800) database system and signaling system 7 (SS7) protocols. The local central office or tandem switch uses the SMS/800 database system to translate the single inbound toll-free number for the business call center into a matching local phone number terminating at the local POP call center gateway. The local central office or tandem switch identifies to the local POP call center gateway the translated called party number and optionally the calling party number. The call is terminated at the local POP call center gateway, which answers the call with an automated voice response system.

The POP call center gateway uses the translated called party number to identify and run a matching interactive voice response application customized to the business call center. The customized interactive voice response application can either be replicated at each local POP call center gateway or dynamically accessed from the business call center just in time as the call is answered and serviced. In either case, the POP call center gateway responds to the incoming call with an interactive voice response application customized to the business call center that was called by the customer. This custom interactive voice response application then services the customer call by providing appropriate prompts and menus, gathering input from the customer and interactively providing automated service. The custom interactive voice response application uses a virtual private network (connecting the POP call center gateways to one or more business premises call center gateways) to securely access the customer application and/or data at the corresponding business call center to appropriately service the calling customer.

If the call is to be held waiting for an available call center agent, the POP call center gateway holds and queues the call locally while requesting the corresponding premises call center gateway to insert a proxy call into the business call center's ACD. The POP call center gateway, optionally plays music and/or periodic prompts and messages to the caller while the call is on hold. The corresponding premises call center gateway inserts a proxy call in the business call center's ACD and starts monitoring its progress through the ACD queue.

When the proxy call reaches the head of the queue and is about to be answered by a live call center agent, the premises call center gateway alerts the waiting POP call center gateway. The waiting POP call center gateway then routes the locally queued call to the premises call center gateway over an appropriate long distance network, either a public/private switched telephone network or a public/private Internet Protocol (IP) telephony network. The corresponding premises call center gateway receives the routed call, matches it with the corresponding proxy call to the business call center ACD and bridges the incoming call to the proxy call.

A selected call center agent at the business call center then answers the call and provides expected customer service to the customer. Finally, when the customer or the call center agent hangs up the call, the appropriate call center gateway detects the event and alerts the matching counter-part gateway and both gateways terminate the call.

SYSTEM DESCRIPTION

FIG. 3 is a functional diagram of a point-of-presence (POP) call center system in accordance with at least one embodiment of the present invention wherein the end user 116 is connected to the POP-enabled business call center 150 via an originating Local PSTN 106, a Long Distance Network 114 and a terminating Local PSTN 106.

The POP call center system extends the conventional premises and network based call center systems to a fully distributed call center system with multiple points of presence. The POP call center system is capable of locally answering, servicing, queuing and routing inbound toll-free calls to business call centers thereby saving on communications costs and increasing operating efficiency.

The POP call center system consists of one or more POP call center gateway servers 146 distributed at one or more points of presence 152 close to the points of call origination. These POP call center gateway servers 146 are connected by one or more call center networks 148 to premises call center gateway servers 142 at one or more POP-enabled business call centers 150. The POP call center gateway server 146 is connected to a Switch 144 enabling it to receive and originate calls on the local PSTN 106. The POP call center gateway servers 146 are further connected to a switched or dedicated access public telecommunications network 114 enabling long distance voice communications with connected premises call center gateway servers.

A POP-enabled business call center 150 consists of one or more premises call center gateway servers 142, one of which would be selected dynamically at the time of handling of an incoming call at a POP call center gateway.

Referring to FIG. 4, a POP call center gateway 166 intercepts and answers inbound toll-free calls at or near their point of call origination. In addition, it provides automated service with interactive voice response applications, holds and queues the calls until appropriate operators are available to service the call, and plays music or customized announcements to the caller while the call is on hold. If a call is queued, this gateway further requests a corresponding premises call center gateway 164 to originate a proxy call at the call center ACD on its behalf and monitor the progress of the queued call. When the premises call center gateway 164 alerts the POP call center gateway 166, the POP call center gateway 166 routes the locally queued call to the premises call center 150 just in time before the operator picks up the call.

The premises call center gateway 164 responds to requests for call center information and applications from POP call center gateways 166, accesses the requested information and applications from premises call center database systems and supplies it to the requesting POP call center gateway 166. The premises call center gateway 164 further receives proxy call origination requests from the POP call center gateways 166 and generates proxy calls on

their behalf to the premises call center automatic call distributor (ACD). The premises call center gateway 164 then monitors the progress of proxy calls within the ACD for operator availability, communicates with the appropriate originating POP call center gateway 166 for just in time call delivery to the selected operator, and bridges the calls between the POP call center gateway 166 and the premises ACD.

Referring to FIG. 5, a call center network according to one embodiment is a virtual private network connecting the POP call center gateways to one or more premises call center gateways all of which belong to a single business call center. A virtual private network offers industry standard connection and transport protocols such as ATM, Frame Relay or Internet Protocol (IP) for secure and private data communications between connecting entities with optional quality of service guarantees. Referring to FIG. 6, each POP call center gateway can be part of multiple such call center networks one for each business call center that it serves. POP call center gateways use a call center network to connect to corresponding premises call center gateways and access appropriate interactive voice response applications and information as well as request proxy call origination and monitoring of call progress. A call center network can optionally support voice communications over ATM, Frame Relay or IP protocols. In such a case, the POP call center gateways can use the call center network as an alternative long distance voice communications network when calls are bridged across the premises call center gateway to the business call center ACD.

Referring to FIG. 7, all the call center networks connect to a global POP call center network directory service 194 for translating the called party number of an incoming call at a POP call center gateway to the network address of a corresponding premises call center gateway. For each called party number at each POP call center, the POP call center network directory maintains a service record containing at a minimum the corresponding premises call center gateway network address. In one preferred embodiment of the invention, a POP call center network directory service uses a network directory based on the Internet standard Lightweight Directory Access Protocol (LDAP).

A POP call center gateway further comprises POP call manager 182, POP voice response client 184 and POP network manager 186 software modules hosted on an industry-standard computer telephony server. A computer telephony server consists of an industry standard server computer such as an Intel PC server or Sun Microsystems server enhanced with telephony and voice processing capabilities and running an industry standard applications server operating system such as Microsoft Windows NT or Sun Microsystems Solaris. In an alternative preferred embodiment, a POP call center gateway can comprise an IP telephony gateway server and a separate applications server connected over a high-speed local area network. An IP telephony gateway is capable of translating traditional circuit switched voice communications to packet switched communications and transporting voice over long distance using IP networks. In such a configuration, the applications server hosts the POP call manager, POP voice response client and POP network manager modules which interact with the IP telephony gateway for voice communications and signaling.

A premises call center gateway further comprises premises call manager 188, premises voice response server 190 and premises network manager 186 software module hosted on an industry standard computer telephony server similar to the one hosting the POP call center gateway. In an alternative preferred embodiment, a premises call center gateway can comprise an IP telephony gateway server and

a separate applications server connected over a high-speed local area network. In such a configuration, the applications server hosts the premises call manager, premises voice response server and premises network manager software modules which interact with the IP telephony gateway for voice communications and signaling.

For each participating business call center network, the POP call center system assigns a unique universally accessible inbound toll-free number. This number can be a previously existing 800/888 toll-free access number of a participating business call center. Depending on the geographic areas in which it wishes to receive POP call center service, the participating business call center chooses one or more POP call centers to be connected to its call center network. The POP call center system then assigns a distinct direct inward dial (DID) number for each POP call center connected to the business call center network. This DID number, also referred to as the POP call center called party number, uniquely identifies at each POP call center the specific business call center to which an incoming call is targeted. The POP call center gateway uses this called party number to identify the network address of the corresponding premises call center gateway.

The POP call center system uses the point of call routing capability of the SMS/800 database management system to route toll-free inbound calls originating in a local PSTN to the nearest and most cost effective POP call center capable of handling these calls. Local exchange carriers use SMS/800 database management system to intelligently route inbound toll-free calls to appropriate inter exchange carriers or other competitive local exchange carriers. For each inbound toll-free call targeted at a toll-free number, the responsible central office switch or tandem switch requests routing instructions from the SMS/800 database management system utilizing Signaling System 7 (SS7) protocols. The local PSTN switch then routes the call to the appropriate carrier based on the response received from the SMS/800 database management system. The POP call center system programs the SMS/800 database management system to instruct the local PSTN switch to route the call to the appropriate POP call center using its uniquely assigned DID number matching the originally called toll-free number.

A POP call center gateway receives and terminates calls originating from a connected local public switched telecommunications network (PSTN) enabling it to locally answer, service and queue the calls. A POP call center gateway can be connected to local PSTN at a central office switch, a tandem switch or a LATA tandem switch depending on local telecommunications traffic patterns and geographic location of the PSTN switches. It should be noted that the type and location of PSTN switch to which POP call center gateway is connected determines the local communication costs and geographic coverage for the inbound toll-free calls. In general, connection at a higher level of the PSTN switch results in broader geographic coverage and higher local communications costs.

EXEMPLARY PSEUDO CODE LISTINGS

Pseudo code listings A-D are appended to and form part of this specification. The listings present pseudo code representations of the interactions between the POP modules and the premises modules, particularly the call manager and voice response components. The functionality of each module is described as a set of messages received by the module from other modules and the actions taken by the module in response to these messages. Each module maintains state through

private data structures that are identified in the pseudo code.

POP Call Manager

Referring to pseudo code listing A, appended hereto, a POP call manager receives an incoming toll-free call through the message INCOMING-CALL and requests the local POP network manager to identify the called party number and locate the business call center to which the call is directed by calling TranslateNumberToAddress. The POP network manager implements the translation. The POP call manager determines whether the corresponding call center is able to receive additional incoming calls by sending the message, ALLOCATE-PROXY-CALL, to the premises call manager. If the corresponding call center is able to receive additional incoming calls, the POP call manager attaches the incoming call on an available voice port and transfers the call to the POP voice response client by calling CreateVoiceResponseClientInstance with the operation parameter, ANSWER. If the corresponding call center is unable to receive further calls, the POP call manager generates a busy signal to the local PSTN. If the POP voice response client transfers the call back to it for queuing through the message QUEUE-CALL, the POP call manager requests the corresponding premises call center gateway to originate a proxy call at the premises ACD on its behalf by sending the message PLACE-PROXY-CALL. Upon completion, the POP call manager places the call on hold by transferring the call to the POP voice response client by calling CreateVoiceResponseClientInstance with the operation parameter, HOLD. When the premises call manager alerts the POP call manager that the call is about to be answered by an operator through the message AGENT-READY, the POP call manager terminates the voice response client by sending it the message, TERMINATE. This results in the voice response client passing control of the call back to the POP call manager through the message TRANSFER-CALL. The POP call manager then routes the call over an appropriate long distance voice communications network to the premises call center gateway by calling PlaceCall, bridging the inbound call with the newly placed call by calling BridgeCall. At any time during the call, if the POP call manager receives a termination message USER-TERMINATION from the POP voice response client, which sends such a message if the user terminates the call, it notifies the premises call manager of the event by sending the message TERMINATE-CALL. The POP call manager may also receive a notification from the premises call manager that the agent has terminated the call, through the message AGENT-TERMINATION. In either case, it performs clean up of the incoming and long distance voice ports, and all state data associated with the incoming call.

POP Voice Response Client

Referring to pseudo code listing B, appended hereto, a POP voice response client receives and responds to instructions from the local POP call manager to answer and service an incoming call to a specified business call center. Based on the instructions as well as the configuration options for the specified business call center, the POP voice response client locates and connects to the premises voice response server on the matching premises call center gateway. Thereafter, the POP voice response client interacts with the corresponding premises voice response server to run an interactive voice response application customized to the business call center. Such interactions include accessing the necessary voice prompts, menus, forms, scripts, data and applications from the premises voice response server. It should be noted that the interactive voice response application can be customized to the specified business center by previously loading all the necessary voice prompts, menus, forms, scripts, and

applications at the POP call center gateway. This approach would require full replication of all business applications at all POP call centers, which is wasteful in utilization of resources and expensive due to operational complexity. Thus, while business applications may be replicated at all POP call centers in certain embodiments, the envisioned approach is to use a distributed voice user interface manager that adapts dynamically to the required interactive application customized to the specified business call center.

A preferred embodiment of the invention features a POP voice user interface manager embedded in the POP voice response client enabling it to dynamically adapt to the specified business call center's interactive voice response application without having to locally store all business call center applications at each POP call center. A POP voice user interface manager dynamically accesses voice prompts, menus, forms, scripts and applications customized to a specified business call center as needed from the corresponding premises voice response server. The POP voice user interface manager and the corresponding premises voice response server use a specialized request/response protocol such as the Internet standard Hyper Text Transfer Protocol to access the distributed resources. Optionally, the POP voice user interface manager and the corresponding premises voice response server may use the Internet standard Hyper Text Markup Language (HTML) or its extensions such as Extended Markup Language (XML) to access conveniently packaged units of information or application across the call center virtual private network. The POP voice response client optionally stores frequently used and rarely modified voice prompts and messages locally in a network cache to improve access efficiency. However, the voice response client can also access voice prompts, messages and other audio files in real-time using Internet streaming protocols such as Real-time Transfer Protocol (RTP) across the call center virtual private network.

The POP call manager creates an instance of the voice response client, and passes an incoming call to it to handle, by specifying the operation parameter ANSWER. This results in the voice response client contacting the voice response server at the premises call center gateway by calling AccessScript, with the parameter ANSWER-SCRIPT, to receive the script to execute. This script is passed to ExecuteScript, which processes and executes the script which typically interacts with the user presenting announcements and menu options and accepts user input via touch tones keys on the phone or speech recognition. Thereafter, when an external event occurs, the voice response client processes the event and takes actions. When user input is received through the message USER-INPUT, the input is decoded by calling ProcessUserInput, which determines what action should be performed in response to the input. This is codified by the returned result, UserRequest, which is passed to the voice response server to process by calling AccessScript. AccessScript returns back to the voice response client a new script to process, which is once again passed to ExecuteScript. If the user input is a request for operator assistance, the script returned by the business call center application running on the premises voice response server instructs the POP voice response client to transfer the call back to the local POP call manager to be queued awaiting availability of a call center agent. The POP voice response client sends the message, QUEUE-CALL, to the POP call manager. The POP call manager, as described earlier, queues the call and requests the corresponding premises call manager to generate a proxy call at the business call center ACD on its behalf. The POP call manager also creates an instance of the voice response client with the operation parameter HOLD. The general operation of the voice response client in this case is identical to the case

of ANSWER, except that the first script, which is requested by calling AccessScript, is with the parameter HOLD-SCRIPT. This parameter serves to distinguish the script executed by the voice response client when the user is on hold from when the user is going through self-service.

When the control events USER-HANGUP, which is generated when a user hangs up the phone, or TERMINATE, which is sent by the POP call manager to terminate user interaction, occur, the voice response client responds by sending the messages USER-TERMINATION and TRANSFER-CALL to the POP call manager.

While the user is awaiting an available agent, the voice response client interacts with the voice response server to acquire status updates regarding the progress of proxy calls that the premises call manager had originated. Based on this information and call center configuration options, the voice response client alerts the waiting caller with status update messages. The script selected by HOLD-SCRIPT controls this behavior.

A POP network manager receives requests from the local POP call manager to translate the called party number of an incoming call to the network address of the premises call center gateway by a call to its procedure TranslateNumberToAddress. The POP network manager in turn requests the global POP network directory service to retrieve the entry corresponding to the specified called party number. The POP network manager accesses the network address of the corresponding premises call center gateway from the retrieved entry and returns it to the requesting local POP call manager. It should be noted that the global POP network directory service could be implemented using a single directory server or a collection of directory servers with replicated data for additional reliability. It should also be noted that the directory servers could be co-located at the POP call centers. As noted earlier, the directory service can be implemented using Internet standard LDAP compliant directory services.

Premises Call Manager

Referring to pseudo code listing C, appended hereto, a premises call manager receives requests from POP call managers to allocate proxy call resources at the business call center ACD on their behalf, originate a proxy call on the allocated resources and finally bridge the incoming call from the POP call manager with the ACD line. The premises call center gateway is equipped with an inbound and an outbound voice port pair. The inbound voice port receives a call from the POP call manager and the outbound port is connected to the ACD, and looks like an incoming voice line to the ACD. When the agent is ready to take the user's call, the inbound and outbound ports are bridged to pass the bi-directional conversation. It should be noted that as far as the call center ACD is concerned, such a call appears no different than if it were to be received on one of its inbound trunks.

A POP call manager requests a premises call manager to allocate a port line pair through the message ALLOCATE-PROXY-CALL. Upon receiving such a request, the premises call manager locally creates a proxy call record and allocates a voice port pair for the call by calling AllocateLinePair. If this allocation is successful, it creates a new proxy call associated with the voice port pair and the incoming call, and returns a handle to the proxy call to the invoking POP call manager. If it is unable to allocate a line pair, it returns failure, causing the invoking POP call manager to produce a busy tone to the caller.

When the POP call manager requests the premises call manager to place the proxy call to the ACD through the message PLACE-PROXY-CALL, the premises call manager places the call to the ACD, noting the ACD's call ID. It returns success to the POP call manager, setting the state of the call. The premises call manager then monitors the progress of the proxy call using the ACD's CTI interface. When the proxy call is about to be delivered to a live call center agent, the call center ACD alerts the premises call manager through the message AGENT-READY. Upon receiving the notification, the premises call manager identifies the POP call manager originally responsible for the proxy call and the matching inbound voice port using the proxy call record. The premises call manager then notifies the responsible POP call manager specifying a direct inward dialing (DID) number corresponding to the inbound voice port. Upon receiving such notification from the premises call manager, the receiving POP call manager identifies the matching queued call and dials the provided DID number to transfer it to the notifying premises call center gateway. When this call is received by the premises gateway, it appears to the premises call manager through the message INCOMING-CALL. Using the inbound voice port on which the call arrives at the premises call center gateway, the premises call manager matches it to the local proxy call record and bridges the call to the corresponding outbound voice port, by calling BridgeCall. This series of operations results in the customer call queued at the POP call center to be connected just in time to the appropriate business call center agent, as the agent becomes available. At any time during the call, if the premises call manager receives the message AGENT-TERMINATION from the ACD, it notifies the POP call manager of the event by sending the message AGENT-TERMINATION. The premises call manager may also receive a notification from the POP call manager that the user has terminated the call, through the message TERMINATE-CALL. In either case, it performs clean up of the voice port pairs and the proxy call record.

The above described techniques can be extended to accommodate multiple premises call center gateways at a single location. This is accomplished through one of the premises call center gateways acting as a master, selecting an appropriate gateway for handling an incoming call and returning that gateway address as part of processing INCOMING-CALL. Similarly, multiple POP-enabled business call center locations, each with one or more premises call center gateways can also be accommodated by one of the locations acting as the master site receiving the incoming call requests.

Voice Response Server

Referring to pseudo code listing D, appended hereto, a premises voice response server hosts interactive voice response applications including voice prompts, menus, scripts and forms customized to the local business call center. The premises voice response server connects to business call center databases 198 to access customer and business information as needed by the hosted interactive voice response applications. In one preferred embodiment, the premises voice response server dynamically down loads all or parts of the requested interactive voice response applications to the requesting POP voice response client. The voice response client requests two types of scripts, one designated by the message ANSWER-SCRIPT and the other by HOLD-SCRIPT. These scripts may perform different functions for the business call center depending on whether the phone is being answered or the call is being placed in a queue awaiting an agent. Thereafter, the voice response client requests through a generic message VRC-REQUEST, which provides sufficient context such as requesting voice response client, last script executed, and new user input. In

an alternative preferred embodiment, the voice response server responds to specialized request protocols such as HTTP from remote voice user interface managers embedded in POP voice response clients distributed at POP call centers. In response to such requests, the premises voice response server supplies the requested voice prompts, menus, forms and scripts to the requesting voice user interface manager. This configuration allows the voice user interface manager embedded in the POP voice response client to adapt dynamically to the interactive voice response application customized to the business call center to which the inbound call is directed.

A premises network manager connects to the call center network for the corresponding business call center. The premises network manager initially registers with the global network directory service and creates a business call center service record for each POP call center connected to its call center network. The service record contains at a minimum the called party number at the POP call center corresponding to its business call center and the matching premises call center gateway network address. As described earlier, the POP call manager accesses this service record to identify the business call center gateway corresponding to an inbound call arriving on a particular called party number.

In a preferred embodiment, the POP call center gateway and the premises call center gateway each use two voice ports to bridge the call between the user and the long distance network, and the long distance network and the ACD. If the POP call center connects to a CTI-enabled switch, the requirement to bridge the call is eliminated in the POP call center gateway, since the POP call manager can request the switch perform a "transfer connect" through the CTI interface. Likewise, if the premises call center utilized a CTI-enabled PBX, the need to bridge the voice call at the premises is similarly eliminated.

SYSTEM MANAGEMENT

Configuration of Call Center Networks

When a business call center network is created, the POP call center network directory service entries are required to be created, as it is the configuration source to the network of all POP call centers. The business selects the participating POPs and this determines the allocation of telephone numbers local to the POP's LATA. The set of all the telephone numbers, along with area code is also registered with the SMS/800 system to enable the point of call routing of the toll-free number. The address of the premises call center gateway server is associated with each of the telephone numbers allocated for the business. This ensures that when the call arrives, the query for the address of the premises call center gateway server is correctly handled. The query is based on the local number allocated at the POP to which the SMS/800 system referred the toll-free call. Also to be registered at the directory service is the starting point Uniform Resource Locator (URL) for any HTTP based communication, such as the POP voice response client.

Management of Audio Media in the POP

The voice response client at the POP call center gateway executes voice response application scripts that are created by the business. It is expected to handle a large number of audio media files in the course of executing these

voice response application scripts. When a business changes its media files the voice response client would download large amounts of data in order to replace the cached audio resources. This process should be performed when not handling a user call. Accordingly, an auxiliary media management process in the POP call center system coordinates the validation and replacement of cached files as a maintenance task within the system.

In the foregoing specification and in the following pseudo code listings which form part of the specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made to the specific exemplary embodiments without departing from the broader spirit and scope of the invention as set forth in the appended claims. Accordingly, the specification and drawings are to be regarded in an illustrative rather than a restrictive sense.

Pseudo Code Listing A - POP Call Manager

POPCallManager (...) [
Messages:

```
[INCOMING-CALL, QUEUE-CALL, TRANSFER-CALL,
AGENT-READY, USER-TERMINATION,
AGENT-TERMINATION];
```

Data:

```
ICR [
POPGatewayAddress;    // Address of the POP CC Gateway
InLineNumber;         // Unique Line number of the line within
local system
CalledPartyNumber;    // Number that was called
CallerNumber;         // Caller's number
PremisesGateway;      // Structure of Addresses of the Premises
Servers hosting        // Call Manager Server & Voice response
server and possibly
others
ProxyCallID;          // Id of the Proxy Call registered with Call
Manager Server
VRCInstance;          // Handle to the associated instance of Voice
Response Client
PremisesNumber;       // Number provided by Call Manager Server
to transfer call
OutLineNumber;        // Unique Line number of a line to use to
transfer call
]                      // Incoming Call Records
```

Program:

```
Initialize( ... );
While ( FOREVER ) {

    Message = ReceiveMessage (...);
    Switch ( Message.Operation ) {
    Case INCOMING-CALL: // Request to process incoming
                        call from the PSTN

        // Incoming Call on "LineNumber", to
        "CalledPartyNumber" from "CallerNumber"
        ICR = CreateIncomingCallRecord ( ... );
        // Set ICR LineNumber, CalledPartyNumber &
        CallerNumber to Call Values ICR.InLineNumber =
        Message.LineNumber;
        ICR.CalledPartyNumber = Message.CalledPartyNumber;
        ICR.CallerNumber = Message.CallerNumber;
        // Translate Called Party Number to authoritative Premises
        Gateway Address
        PremisesGateway = TranslateNumberToAddress (
        ICR.CalledPartyNumber, ... );
        // Send Proxy Call Request to Premises Call Manager at
        PremisesGateway,
        // passing Called Party Number and Caller Number.
        PremisesGateway may return
        // the address of a different (available) gateway.
        ProxyCall = SendMessage
        ( PremisesGateway.PremisesCallManager,
        ALLOCATE-PROXY-CALL, .... );
        // If Proxy Call request is successful, accept the call being
        presented on Line Number
        // Record the returned ID & Gateway address for
        subsequent communication.
        If ( ProxyCall.Reply = SUCCESS ) {

            ICR.ProxyCallID = ProxyCall.ID;
            ICR.PremisesGateway = ProxyCall.PremisesGateway;
            // Create a new Voice Response Client instance and
            instruct it to answer call
            ICR.VRCInstance = CreateVoiceResponseClientInstance
            ( ICR, ANSWER, ... );
            ...

        } Else {

            // Respond with Number Busy on Call being Presented
            on LineID
            ...

        }
    }
    Break;
```

Case QUEUE-CALL: // Request to queue a call at Premises
ACD

// Call is identified by I

Case AGENT-READY: // Notification that agent is ready
to receive call

```
// Call is identified by Proxy call identifier
ICR = FindICRFromProxyCallID ( Message.ProxyCallID,
... );
// Premises Call Manager sends the appropriate Premises
Number to call
ICR.PremisesNumber = Message.PremisesNumber;
// Request Voice Response Client Instance to relinquish call
SendMessage ( ICR.VRCInstance, TERMINATE, ... );
...
Break;
```

Case TRANSFER-CALL: // Request to transfer a call to
Premises Call Manager

```
// Call is identified by ICR
ICR = Message.ICR;
// Allocate an available outbound line
ICR.OutLineNumber = AllocateLine ( );
// Place a call on allocated line to Premises Call Manager
using Premises Number
PlaceCall (ICR.OutLineNumber, ICR.PremisesNumber,
... );
// When the call is accepted bridge the incoming call to
outbound call
BridgeCall (ICR.InLineNumber, ICR.OutLineNumber, ... );
...
Break;
```

Case USER-TERMINATION: // Notification that
a user has terminated a call

```
// Call is identified by ICR
ICR = Message.ICR;
// Inform Premises Call Manager of user termination event
SendMessage(ICR.PremisesGateway.PremisesCallManager,
TERMINATE-CALL,
```

```
ICR.ProxyCallID, ... );
```



```

// Terminate call and clean up
TerminateCall ( ICR.InLineNumber, ICR.OutLineNumber,
... );
DeleteIncomingCallRecord ( ICR );
CleanUp ( );
...
Break;

```

Case AGENT-TERMINATION: // Notification that an agent has terminated a call

```

// Call is identified Proxy call identifier
ICR = FindICRFromProxyCallID ( Message.ProxyCallID,
... );
// Terminate the inbound and outbound calls and clean up
TerminateCall ( ICR.InLineNumber, ICR.OutLineNumber,
... );
DeleteIncomingCallRecord ( ICR);
CleanUp ( );
...
Break;

```

```

}

```

```

} // End POPCallManager

```

Pseudo Code Listing B - Voice Response Client

```

VRCInstance ( ICR, Operation, ... ) [
Messages:

```

```

[USER-INPUT, USER-HANGUP, TERMINATE];

```

Data:

```

ICR [ ... ];
Operation: [ANSWER, HOLD];

```

Program:

```

Initialize ( ... );
// Access the Voice Response Server at Premises Gateway for

```

Pat. No. 6011844, *

```

Starting Script
VoiceResponseServer =
ICR.PremisesGateway.VoiceResponseServer;
If (Operation = ANSWER) [

    // If invoked to answer the call, access the answering script from
    Voice Response Server
    NewScript = AccessScript (VoiceResponseServer,
    ANSWER-SCRIPT, ... );

] Else [

    // If invoked to hold the call, access the hold script from Voice
    Response Server
    NewScript = AccessScript (VoiceResponseServer,
    HOLD-SCRIPT, ... );

]
// Execute the accessed script on appropriate line
Status = ExecuteScript ( ICR.LineNumber, NewScript, ... );
// If the call needs to be queued, inform tbe POP Call Manager
If ( Status = QUEUE-CALL ) [

    SendMessage ( POPCallManager, QUEUE-CALL, ... );
    CleanUp ( );
    Exit ( ... );

]
While ( FOREVER ) [

    Message = Receive.Message ( ... );
    Switch ( Message.Operation ) [
    Case USER-INPUT: // User input in the form of touch
    tones or speech recognition

        // Process user input and access additional scripts from Voice
        Response Server, if needed
        UserRequest = ProcessUserInput (Message, ... );
        NewScript = AccessScript (VoiceResponseServer,
        UserRequest, ... );
        // Execute the accessed script on appropriate line
        Status = ExecuteScript ( ICR.LineNumber, NewScript, ... );
        // If the call needs to be queued, inform the POP Call Manager
        If ( Status = QUEUE-CALL) [

            SendMessage ( POPCallManager, QUEUE CALL, ICR,
            ... ); CleanUp ( );

```

Pat. No. 6011844, *

```
Exit ( ... );
```

```
]
...
Break;
```

```
Case USER-HANGUP: // User hangs up the call
```

```
// Inform the POP Call Manager that the user has terminated
the call
SendMessage ( POPCallManager, USER-TERMINATION,
ICR, ... );
// Clean up and exit
CleanUp ( ... );
Exit ( ... );
```

```
Case TERMINATE: // POP Call Manager terminates hold
session
```

```
// Transfer call back to POP Call Manager
SendMessage ( POPCallManager, TRANSFER-CALL, ICR,
... );
// Clean up and exit
CleanUp( ... );
Exit ( ... );
```

```
]
```

```
]
```

```
] // End VRCInstance
```

Pseudo Code Listing C - Premises Call Manager

```
PremisesCallManager ( ... ) [
Messages:
```

```
[ALLOCATE-PROXY-CALL, PLACE-PROXY-CALL,
AGENT-READY, INCOMING-CALL, USER-TERMINATION,
AGENT-TERMINATION];
```

```
Data:
```

PCR [

```

POPGateway;          // Address of the POP Gateway responsible
for this call

```

```

CalledPartyNumber;    // Number that was originally called by
the user

```

```

CallerNumber;         // Caller's number

```

```

// Total number of Line pairs limit the maximum number of
active calls at all POPs
LinePair {

```

```

    InLineNumber;      // Line number on which POP Gateway
call arrives
    OutLineNumber;     // Line number on which ACD call is
placed

```

```

};
ProxyCallStatus;      // Lines are allocated in pairs.
Manager              // Status of the proxy call in Premises Call
ProxyCallID,          // ID of the proxy call that is provided to the
POP call manager
ACDCallStatus;        // Status of the proxy call in the ACD

```

```

ACDInLineNumber:      // ACD Line number on which
outbound call is placed
ACDCallID;            // ACD Call handle

```

```

] // Proxy Call Record

```

Program:

```

Initialize ( ... );
While ( FOREVER ) [

```

```

    Message = ReceiveMessage ( ... );
    Switch ( Message.Operation ) [
    Case ALLOCATE-PROXY-CALL: // Request from a POP
for a proxy call allocation

```

```

        // Allocate a line pair. There must be as many line pairs as
there are inbound ACD ports

```

```

Pat. No. 6011844, *
// If successful, then create proxy call record and assign the
line pair to it. AllocateLinePair
// could be extended to return the address of an available
premises call center
// gateway if this gateway does not have available line pairs.
Status = AllocateLinePair ( LinePair, ... );
If ( Status = SUCCESS ) {

    // Proxy call identifier is assigned to the proxy call record
    PCR = CreateProxyCallRecord ( ... );
    // Assign allocated line pair to proxy call record
    PCR.LinePair = LinePair;
    // Assign POP Gateway address, Called Party Number and
    Caller Number to Proxy
    // Call record
    PCR.POPGateway = Message.PopGateway;
    PCR.CalledPartyNumber = Message.CalledPartyNumber;
    PCR.CallerNumber = Message.CallerNumber;
    PCR.ProxyCallID = &PCR; // address handle to PCR
    // Reply to POP Call Manager indicating success and pass
    Proxy call identifier
    SendMessage ( PCR.POPGateway.POPCallManager,

        SUCCESS, PCR.ProxyCallID, ... );

    PCR.ProxyCallStatus = ALLOCATED;

} Else { // Reply to POP Call Manager indicating rejection of
incoming call

    SendMessage ( Message.umber
    PCR.ACDCallID = ProxyCall.ACDCallID;
    PCR.ACDInLineNumber = ProxyCall.ACDInLineNumber;
    // Reply to POP Call Manager indicating successful queuing
    of call at ACD
    SendMessage ( PCR.POPGateway.POPCallManager;
    SUCCESS, PCR.ProxyCallID, ... );
    PCR.ProxyCallStatus = QUEUED;
    ...
    Break;

Case AGENT-READY: // Notification from ACD that
the agent is ready to take call

// Call is identified by PCR which is identified by ACD
inbound line number
PCR = FindPCRFromACDInLineNumber (
Message.ACDInLineNumber, ... );
// Translate inbound line number to phone number to be called

```

```

Pat. No. 6011844, *
NewScript = AccessScript ( ANSWER-SCRIPT, ... );
SendMessage ( VRCInstance, NewScript, ... );
...
Break;

```

Case HOLD-SCRIPT: // Request for starting script for servicing a call during hold

```

// Voice Response Client instance is identified by message
VRCInstance = Message.VRCInstance;
// Access the starting script for servicing a call on hold and
send it to VRC instance NewScript = AccessScript
( HOLD-SCRIPT, ... );
SendMessage ( VRCInstance, NewScript, ... );
...
Break;

```

Case VRC-REQUEST; // Request for a script based on current user interaction

```

// Voice Response Client instance is identified by message
VRCInstance = Message.VRCInstance;
// Access the script for servicing the user request and send it
to
VRC instance
NewScript = AccessScript ( Message.UserRequest, ... );
SendMessage ( VRCInstance, NewScript, ... );
...
Break;

```

```

]

```

```

]

```

```

] // End VoiceResponseServer

```

CLAIMS: What is claimed is:

[*1] 1. A method of handling a toll free call that is directed to a remote call center, the method comprising:

redirecting the toll free call from the remote call center to a local call center;

automatically answering the redirected toll free call in the local call center to determine whether connection to the remote call center is necessary;

and

bridging the redirected toll free call with a telephone connection in the remote call center via a long distance network adaptable to be coupled between the local call center and the remote call center if connection to the remote call center is necessary.

[*2] 2. The method of claim 1 further comprising signaling the remote call center via a data network between the local call center and the remote call center to request the telephone connection to be established in the remote call center if connection to the remote call center is necessary.

[*3] 3. The method of claim 1 wherein redirecting the toll free call from the remote call center to the local call center comprises translating a toll free number associated with the inbound toll free call to a translated number that terminates at the local call center.

[*4] 4. The method of claim 1 wherein bridging the redirected toll free call with the telephone connection in the remote call center comprises:

determining when the telephone connection in the remote call center is imminent; and

forwarding the redirected toll free call to the remote call center via the long distance network to be bridged with the telephone connection in the remote call center in response to determining that the telephone connection in the remote call center is imminent.

[*5] 5. The method of claim 4 wherein determining when the telephone connection in the remote call center is imminent comprises the local call center receiving a signal from the remote call center via a data network coupled between the local call center and the remote call center, the signal indicating that the telephone connection in the remote call center is imminent.

[*6] 6. The method of claim 4 further comprising storing an entry in a queue in the remote call center to indicate the request for the telephone connection to be established in the remote call center, and wherein determining when the telephone connection in the remote call center is imminent comprises determining when the entry in the queue in the remote call center has advanced to the head of the queue.

[*7] 7. The method of claim 1 wherein automatically answering the redirected toll free call in the local call center to determine whether connection to the remote call center is necessary comprises:

automatically answering the call using an automated call answering system; and

executing an interactive application in the automated call answering system to interact with a caller.

[*8] 8. The method of claim 7 further comprising downloading at least a portion of the interactive application from the remote call center to the local call center via a data network coupled between the local call center and the remote call center.

[*9] 9. The method of claim 1 wherein bridging the redirected toll free call with the telephone connection in the remote call center via a long distance network comprises bridging the redirected toll free call with the telephone connection in the remote call center via a voice communication channel established over a data network coupled between the local call center and the remote call center.

[*10] 10. A distributed toll free call servicing system comprising:

a remote call center for servicing toll free calls;

a local call center to automatically answer toll free calls that have been redirected from the remote call center to the local call center, the local call center being configured to determine, for each redirected toll free call, whether connection to the remote call center is necessary and, if connection to the remote call center is necessary, to bridge the redirected toll free call with a telephone connection in the remote call center via a long distance network coupled between the local call center and the remote call center.

[*11] 11. The system of claim 10 further comprising a data network coupled between the local call center and the remote call center, the local call center being further configured to signal the remote call center via the data network to request the telephone connection to be established in the remote call center if connection to the remote call center is necessary.

[*12] 12. The system of claim 11 wherein the data network is a virtual private network that provides industry standard connection and transport protocols.

[*13] 13. The system of claim 11 wherein the data network between the local call center and the remote call center forms the long distance network used to bridge the redirected toll free call with the telephone connection in the remote call center.

[*14] 14. The system of claim 11 further comprising a database of network addresses coupled to the local call center via the data network, the local call center being configured to access the database of network addresses to determine a network address of the remote call center in response to answering a toll free call that has been redirected from the remote call center to the local call center.

[*15] 15. The system of claim 14 wherein the local call center indexes the database of network addresses based on a phone number generated by translating a toll free number assigned to the remote call center.

[*16] 16. The system of claim 15 wherein the phone number generated by translating the toll free number assigned to the remote call center is a phone number assigned to the local call center.

[*17] 17. The system of claim 10 wherein the local call center includes an automated call answering system that executes an interactive application to interact with respective callers of the redirected toll free calls.

[*18] 18. The system of claim 17 further comprising a data network coupled between the local call center and the remote call center, the local call

center being further configured to download at least a portion of the interactive application from the remote call center via the data network.

[*19] 19. A distributed toll free call servicing system comprising:

a remote call center for servicing toll free calls;

a plurality of local call centers that are distributed within respective service regions to automatically answer toll free calls that originate within the service regions, each of the toll free calls being redirected from the remote call center to one of the plurality of local call centers selected according to the service region from which the toll free call originated, the selected one of the plurality of local call centers being configured to determine, for each redirected toll free call received, whether connection to the remote call center is necessary and, if connection to the remote call center is necessary, to bridge the redirected toll free call with a telephone connection in the remote call center via a long distance network between the local call center and the remote call center.

[*20] 20. The system of claim 19 further comprising a data network interconnecting each of the plurality of local call centers with the remote call center, the selected one of the plurality of local call centers being further configured to signal the remote call center via the data network to request the telephone connection to be established in the remote call center if connection to the remote call center is necessary.

[*21] 21. The system of claim 20 wherein the data network is a virtual private network that provides industry standard connection and transport protocols.

[*22] 22. The system of claim 20 wherein the data network interconnecting the plurality of local call centers with the remote call center forms the long distance network used to bridge the redirected toll free call with the telephone connection in the remote call center.

[*23] 23. The system of claim 19 wherein each of the plurality of local call centers includes an automated call answering system that executes an interactive application to interact with respective callers of the redirected toll free calls.

[*24] 24. The system of claim 23 further comprising a data network interconnecting each of the plurality of local call centers with the remote call center, at least one of the plurality of local call centers being configured to download at least a portion of the interactive application from the remote call center via the data network.

[*25] 25. The system of claim 19 wherein the service regions correspond to geographic regions.

[*26] 26. The system of claim 19 wherein the service regions are local telephone network service regions.

[*27] 27. A telephone call handling system comprising:

a telephony switch to receive toll free calls that have been redirected to the call handling system by a local switched telephone network;

a computer telephony server coupled to the telephony switch to detect when a redirected toll free call is received in the telephony switch and to automatically answer the redirected toll free call to determine whether connection to a remote call center is necessary, the computer telephony server being configured to bridge the redirected toll free call with a telephone connection in the remote call center via a long distance network adaptable to be coupled between the local call center and the remote call center if connection to the remote call center is necessary.

[*28] 28. The call handling system of claim 27 wherein the long distance network is a data network.

[*29] 29. The call handling system of claim 27 wherein the computer telephony server is configured to issue a request to the remote call center to initiate a proxy call in the remote call center, the proxy call being a request for connection to a human operator that is managed within the remote call center without a long distance voice connection being established between the remote call center and the call handling system.

[*30] 30. The call handling system of claim 29 wherein the computer telephony server is further configured to receive communications via a data network indicating progress of the proxy call in the remote call center and wherein the computer telephony server is further configured to bridge the redirected toll free call with a telephone connection in the remote call center by bridging the redirected toll free call with the proxy call in response to detecting that the proxy call is about to be answered by a human operator.

U.S. Patent

Jan. 4, 2000

Sheet 1 of 7

6,011,844

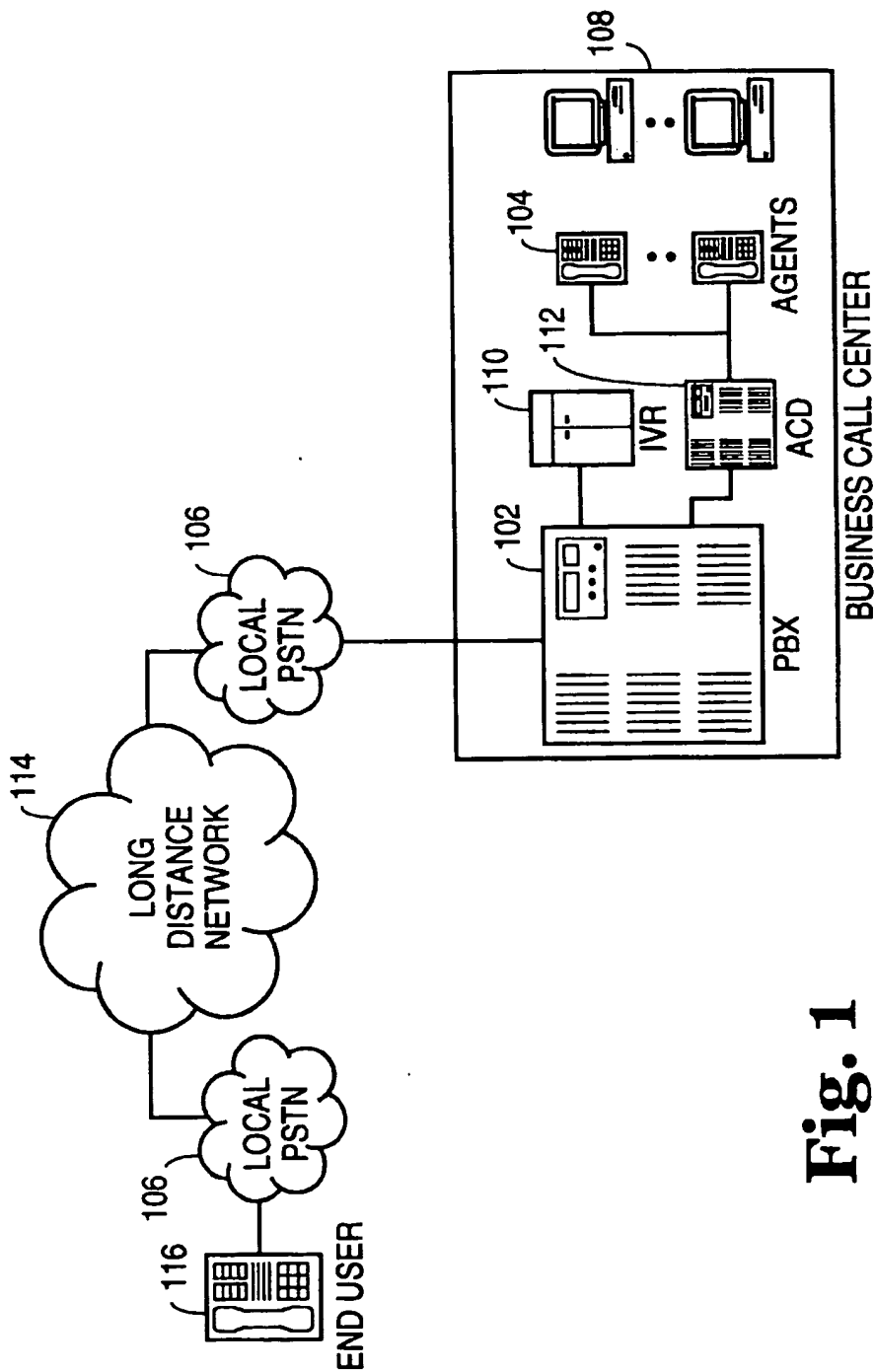


Fig. 1

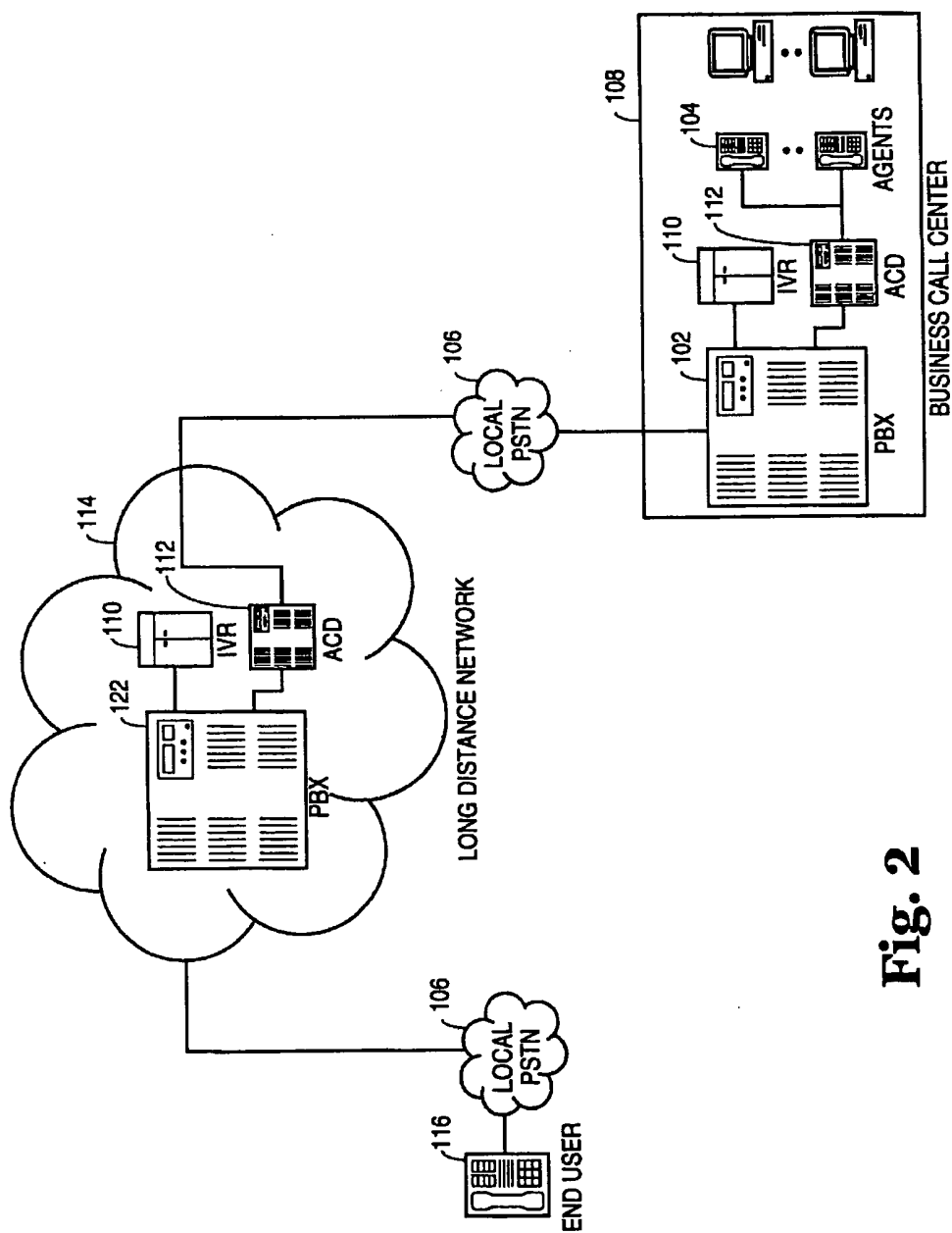


Fig. 2

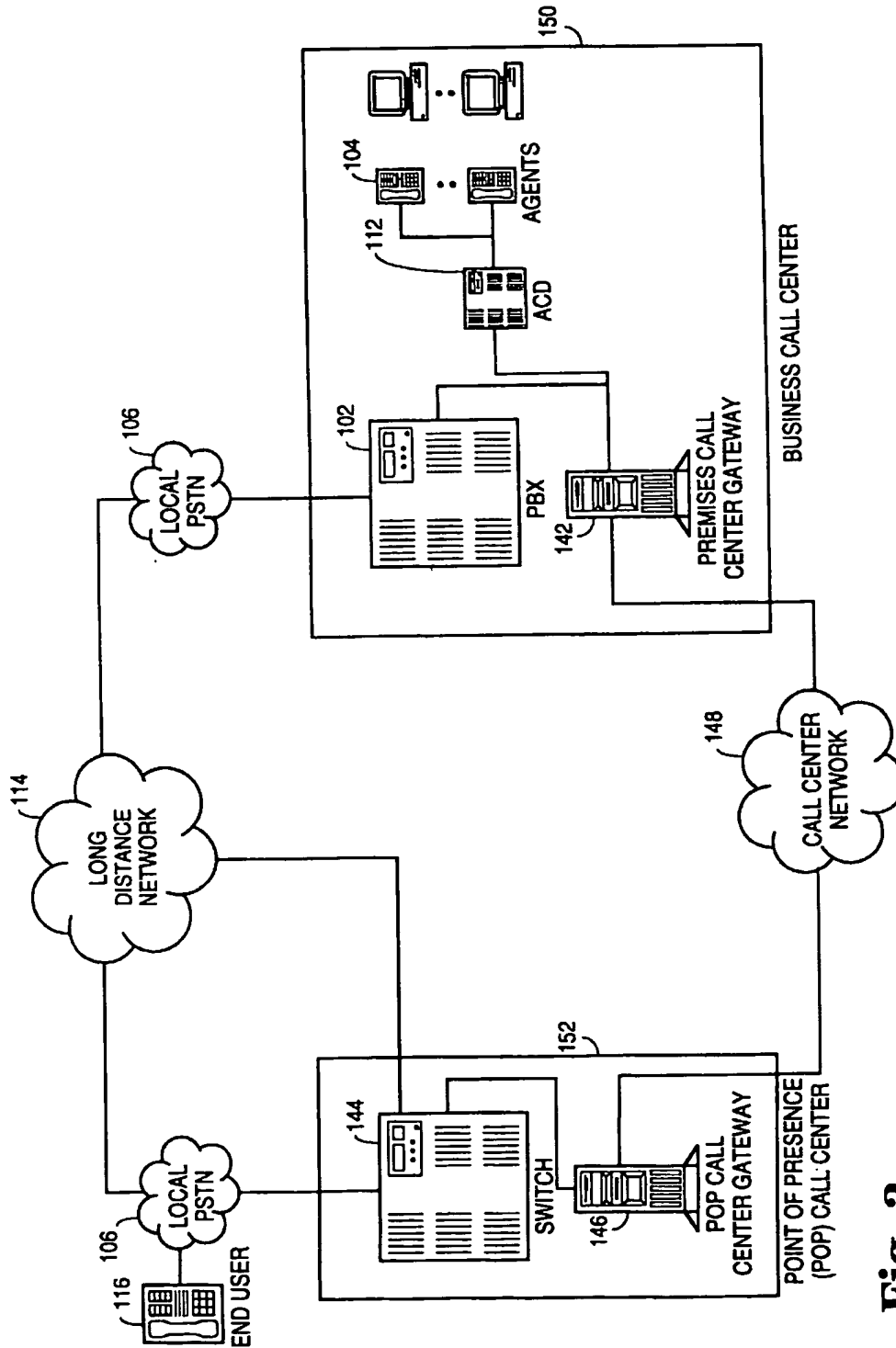


Fig. 3

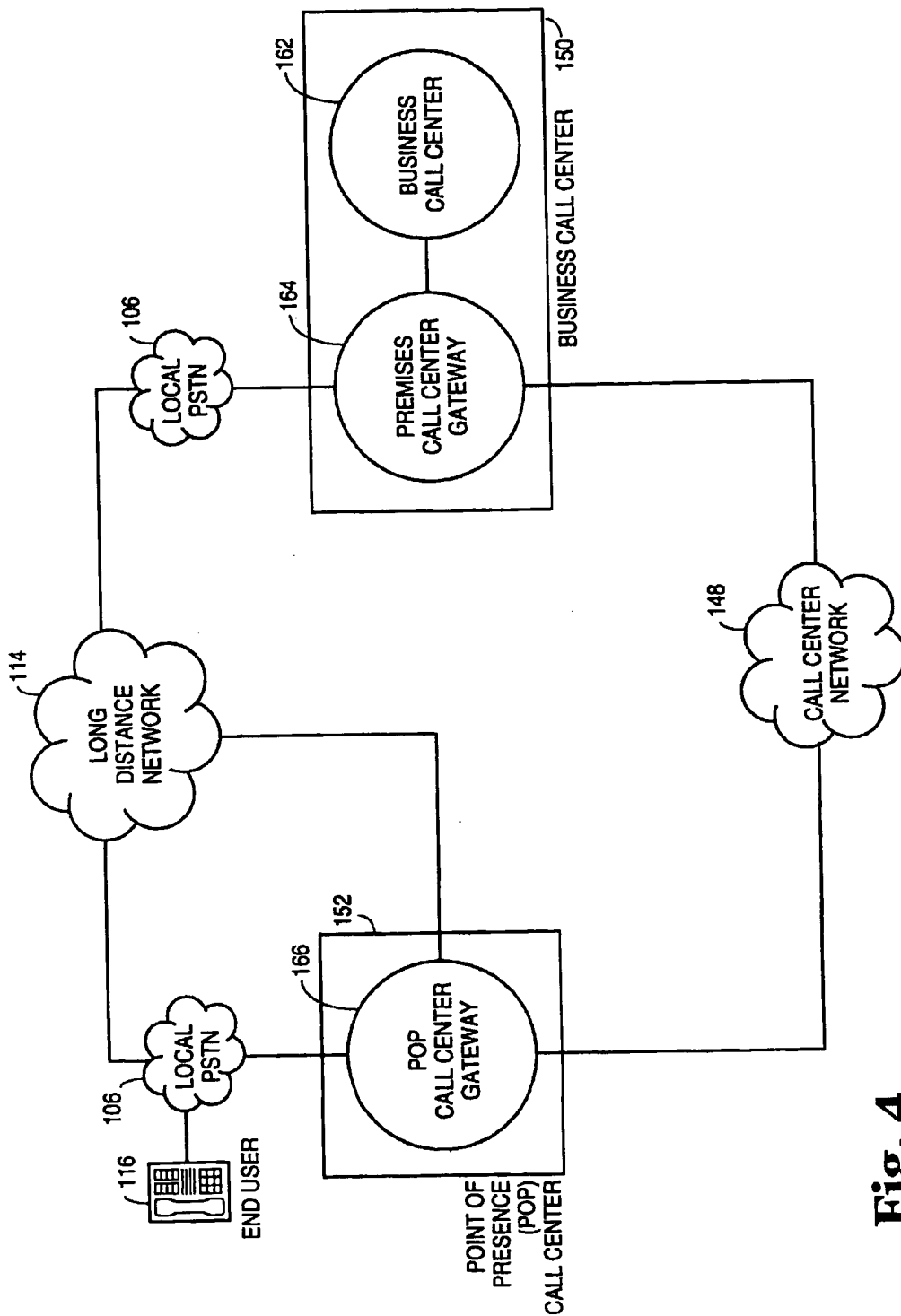


Fig. 4

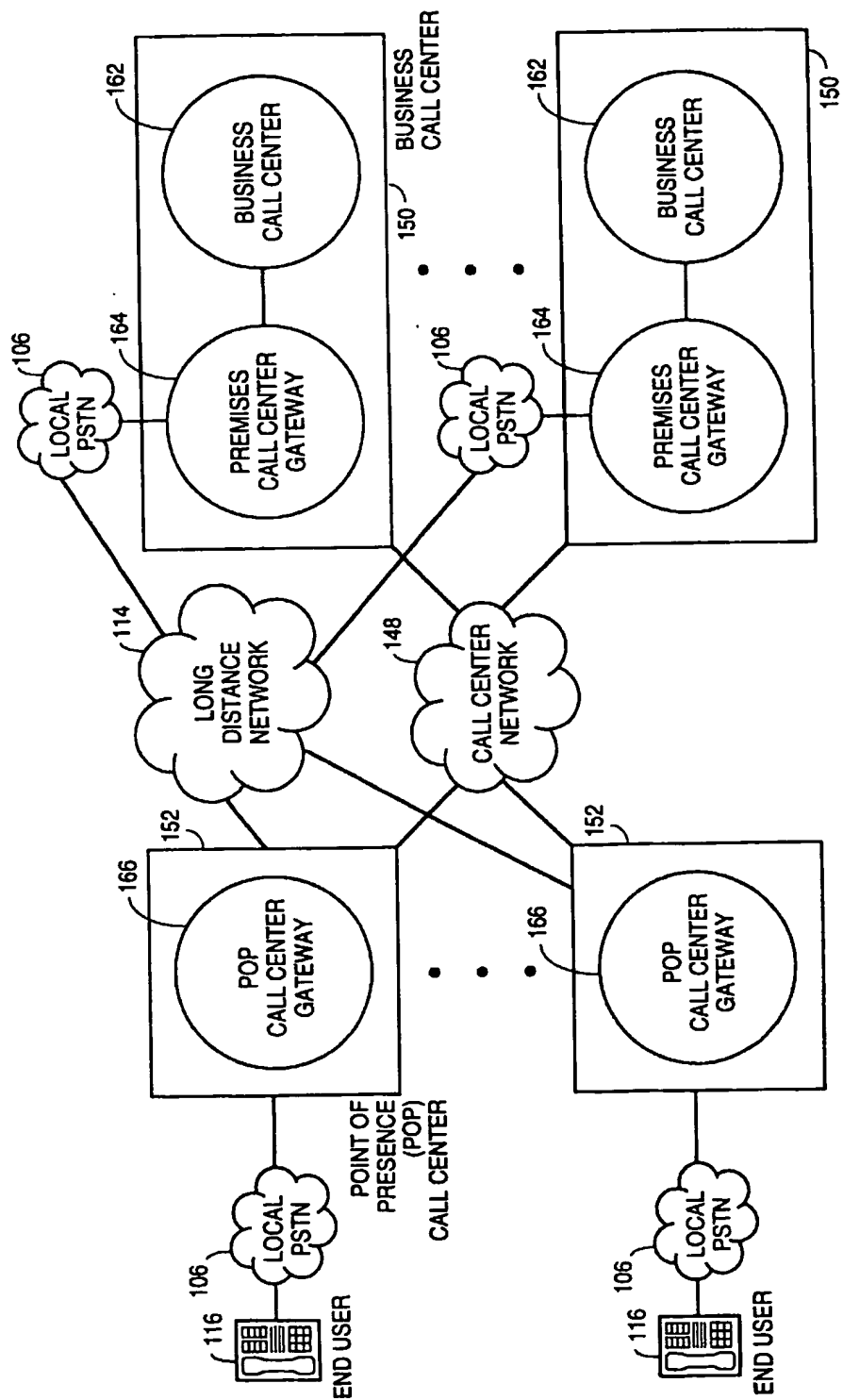


Fig. 5

U.S. Patent

Jan. 4, 2000

Sheet 6 of 7

6,011,844

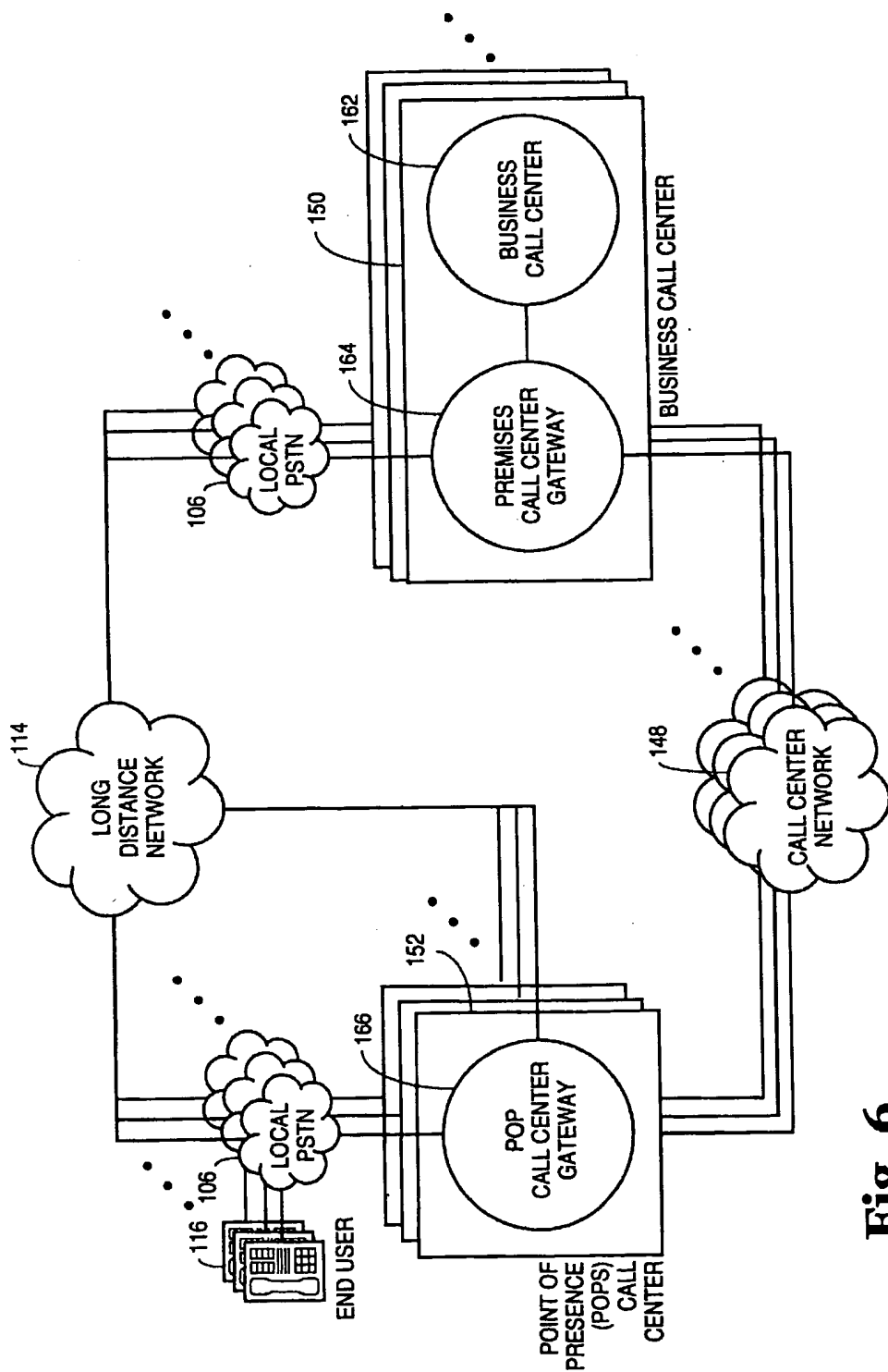


Fig. 6

U.S. Patent

Jan. 4, 2000

Sheet 7 of 7

6,011,844

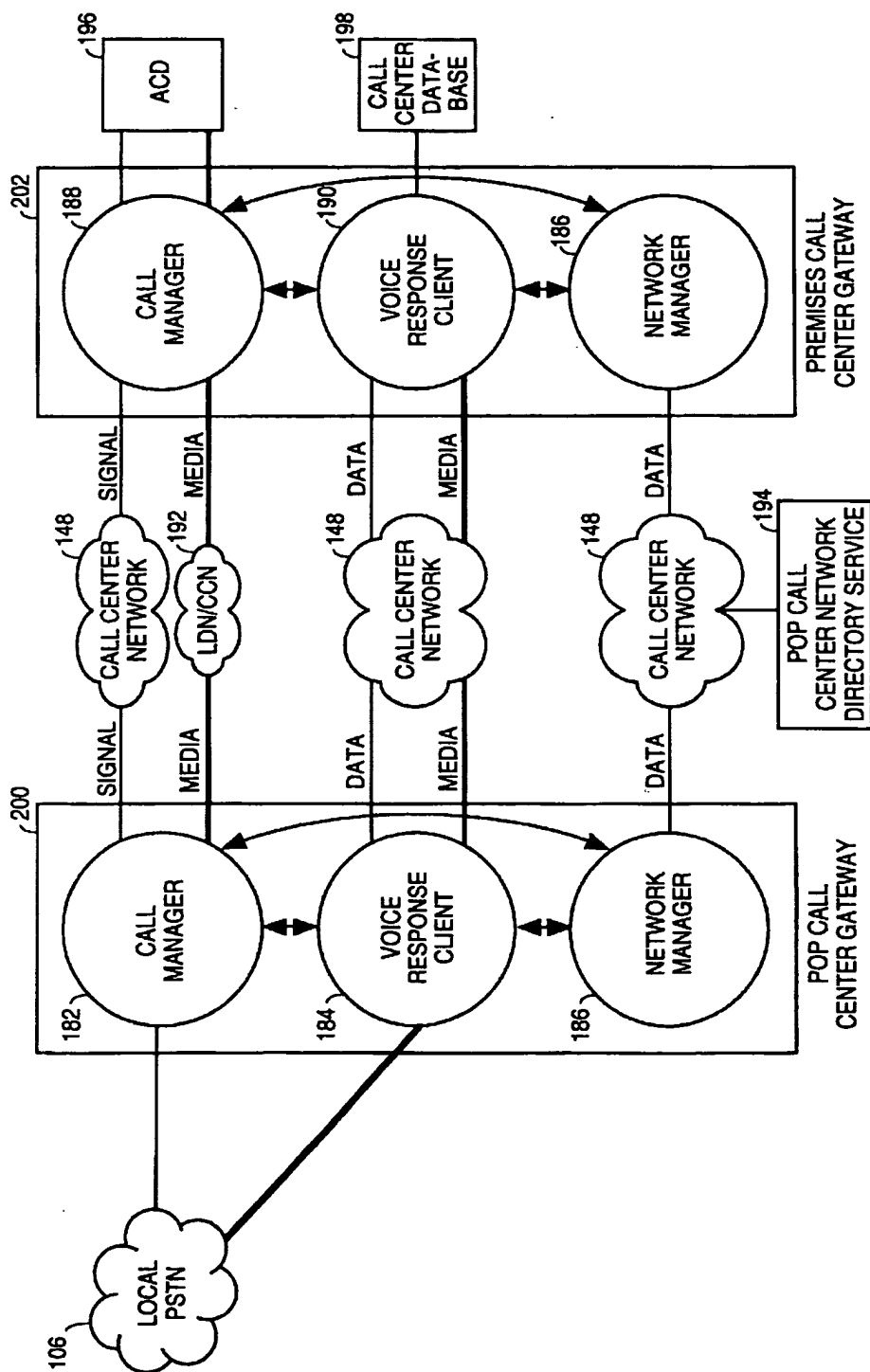


Fig. 7

THIS PAGE BLANK (USPTO)

Pat. No. 5243643 printed in FULL format.

5,243,643

<=2> GET 1st DRAWING SHEET OF 7

Sep. 7, 1993

Voice processing system with configurable caller interfaces

LIT-REEX: NOTICE OF LITIGATION

Intervoice Inc. v. Precision Systems Inc., Filed Apr. 10, 1996, D.C. N.D.
Texas, Doc. No. 3 96cv 994

INVENTOR: Sattar, Sohail, Irving, Texas
Polisky, Steven E., Irving, Texas

ASSIGNEE-AT-ISSUE: Voiceples Corporation, Irving, Texas (02)

ASSIGNEE-AFTER-ISSUE: Date Transaction Recorded: Jul. 31, 1995
ASSIGNMENT OF ASSIGNOR'S INTEREST (SEE DOCUMENT FOR DETAILS).
INTERVOICE, INC., A TEXAS CORPORATION 17811 WATERVIEW PARKWAY DALLAS, TEXAS
75252
Reel & Frame Number: 007570/0451

Date Transaction Recorded: Jun. 07, 1999
ASSIGNMENT OF ASSIGNOR'S INTEREST (SEE DOCUMENT FOR DETAILS).
INTERVOICE LIMITED PARTNERSHIP 639 ISBELL ROAD, SUITE 390 RENO, NEVADA 89509
Reel & Frame Number: 009996/0962

APPL-N0: 774,202

FILED: Oct. 10, 1991

CERTCORR: Mar. 07, 1995 a Certificate of Correction was issued for this Patent

REL-US-DATA:
Continuation-in-part of Ser. No. 608,147, Nov. 1, 1990

INT-CL: [5] H04M 1#57; H04M 1#64

US-CL: 379#88.23; 379#88.18; 379#88.22; 379#93.26; 379#142; 379#201;

CL: 379;

SEARCH-FLD: 379#88, 89, 67, 201, 142, 97

REF-CITED:

U.S. PATENT DOCUMENTS		
4,747,127	5/1988	* Hansen et al. 379#94
4,799,144	1/1989	* Parruck et al. 364#200
4,852,149	7/1989	* Zurick et al. 379#67
4,879,743	11/1989	* Burke et al. 379#142
4,903,289	2/1990	* Hashimoto 379#61
4,930,150	5/1990	* Katz 379#93

		Pat. No. 5243643, *	
4,942,598	7/1990	* Davis	379#57
4,959,854	9/1990	* Cave et al.	379#157
4,985,913	1/1991	* Shalom et al.	379#76
4,996,704	2/1991	* Brunson	379#67
5,003,577	3/1991	* Ertz et al.	379#89
5,036,533	7/1991	* Carter et al.	379#59
5,109,405	4/1992	* Morganstein	379#89
5,128,984	7/1992	* Katz	379#92

OTHER PUBLICATIONS

"Voice Processing Update", Teleconnect, Sep. 1989, pp. 98-138 (ads omitted).

PRIM-EXMR: Brown, Thomas W.

LEGAL-REP: Baker & Botts

CORE TERMS: vector, caller, processing, interface, window, message, user, mail, server, telecommunication, customer, database, computer, input, segment, telephone, subscriber, envelope, stored, exemplary, profile, digit, stack, vendor, integrated, relational, integration, label, accessed, configure

ABST:

A voice mail processing system is provided which comprises a digital computer common-platform operable to communicate through telecommunication lines with an outside environment. Communications between callers and the voice mail processing system are performed through specific caller interfaces that are configurable so as to allow different caller interfaces to each caller to the voice mail processing system. Subscriber profile records that contain caller interface configuration information for each caller can be edited to store desired caller interface configuration information for any caller. Information in any subscriber's profile record is accessed by reference to a caller identification number associated with each caller, generated either by the system or by caller input.

NO-OF-CLAIMS: 15

EXMPL-CLAIM: <=17> 5

NO-OF-FIGURES: 20

NO-DRWNG-PP: 7

PARCASE:

This application is a continuation-in-part of copending U.S. patent application Ser. No. 07,608,147, filed Nov. 1, 1990, and entitled "INTEGRATED VOICE PROCESSING SYSTEM".

SUM:

TECHNICAL FIELD OF THE INVENTION

The invention relates in general to systems and methods for performing voice processing functions, and more particularly to systems and methods of providing multiple and flexible voice mail and integrated voice processing interfaces.

BACKGROUND OF THE INVENTION

Computer-based telecommunications systems have proliferated in the last few years along with the common proliferation of high-speed personal computers and the generally lower costs of equipment now available for use in complex telecommunications applications. With the use of high-speed telephone switching lines, telecommunications applications are exhibiting rapid advancements in technology and versatility. One of the areas in which telecommunications has experienced rapid advancements is the "voice processing" industry, wherein telephone lines provide communication links between users calling in to obtain information from a computer-based system that is adapted to provide information about a particular business or organization.

The voice processing industry provides "voice-based" systems which interact in varying degrees with users seeking information from the system. Voice-based systems have evolved over the last several years into discrete systems which accomplish specific tasks. Thus, the voice processing industry is broken up into a series of "sub-technologies" which are occupied by particular providers and which are further segregated according to the products and services available in the specific sub-technology area. Generally, the voice processing industry has developed the following sub-technology areas: voice messaging ("VM") technology, call processing ("CP") technology, interactive voice response ("IVR") technology, and a number of other limited technologies which at the present are not large and do not command significant market shares, such as for example, the "FAX voice response" technology area. VM systems automatically answer calls and act as "automated attendants" to direct the calls to the proper person or department in an organization. These systems have in the past usually comprised look-up databases that perform voice functions for the user as the user accesses the system. VM technology can be adapted to read electronic mail to a user or caller on a telephone, and may also provide means for storing incoming facsimile messages for forwarding these messages over TOUCHTONE telephones when so instructed. Systems that fall under the VM category may also be adapted to recognize spoken phrases and convert them into system usable data.

Previous VM systems are exemplified in U.S. Pat. No. 4,585,906, Matthews et al. The Matthews et al. patent discloses an electronic voice messaging system which is connected with a user's telephone communications network or private branch exchange (PBX) to provide VM functions for the user. See Matthews et al., col. 4, lines 49-66.

Another example of a VM system is disclosed in U.S. Pat. No. 4,926,462, Ladd et al. The device of the Ladd et al. patent provides methods of handling calls in a VM system based on information supplied by a PBX. See Ladd et al., col. 4, lines 50-52. VM systems taught in the Ladd et al. patent comprise a feature phone emulator interface which emulates known PBX compatible feature phones having multiple line capability. The feature phone emulator is interfaced to the PBX as an actual feature phone, and the PBX is configured to assign a group of extension numbers to line appearances on the feature phone. The VM systems disclosed in the Ladd et al. patent answer the calls to these extensions by using the feature phone emulator interface. See Ladd et al., col. 4, lines 53-65.

Yet another VM system is disclosed in U.S. Pat. No. 4,811,381, Woo et al. The Woo et al. patent discloses a VM system which is connected to a trunk side of a PBX in a business telephone system. The VM system described in Woo et al. provides the feature of answering forwarded calls with a personal greeting from the party whose phone is accessed by a user. See Woo et al., col. 2, lines 37 through col. 2, lines 40-54.

If on the other hand a customer requires a voice processing system to perform on-line transaction processing and interact with a called to answer routine questions about the status of an account, for example, the customer's requirements are usually best addressed by an IVR system which can be viewed as fulfilling requirements presented by a totally different set of architectural problems. Essentially, in an IVR system the user desires to talk to a central processing unit (CPU) to obtain database information. IVRs are particularly useful in the banking industry wherein account holders can call a CPU to get account balances and other relevant information. Generally, IVR systems must also interface to a TOUCHTONE telephone to allow the caller to provide meaningful data to the IVR system which then can return meaningful information to the user.

When a retail company wishes to sell large volumes of merchandise through a "call-in ordering" system, it requires a call processing (CP) system. In the past, before CP systems were available, such retail companies utilized "agents" to handle incoming calls. The agents typically manned a switchboard that allowed manual input of user orders to an ordering system which could have been computer-based. CP technology today provides automatic call distribution ("ACD") which allows a company to nearly eliminate the need for live agents handling phone calls, and replaces the agent with an interactive telephone system through which products can be ordered. The products can be paid for by credit cards having credit card numbers which are input through a TOUCHTONE telephone to a computer ordering system for billing purposes.

Other examples of CP technology are taught in U.S. Pat. No. 4,850,012, Mehta et al. The Mehta et al. patent discloses a CP system for intercepting incoming calls to a key telephone system, and returning a message to a calling party. See Mehta et al., col. 2, lines 11-17. The Mehta et al. system further provides an intercom line for providing voice announcements or messages through the key telephone system to the called parties. CP systems described in Mehta et al. comprise a call processor which intercepts telephone calls wherein an instructional message is returned to the calling party, thereby informing the calling party to select a party associated with the key telephone system by dialing a pseudo-extension number associated with each party. See Mehta et al., col. 2, lines 18-28.

Other technologies have been developed to provide the particular services and solutions to other niches and sub-technologies in the voice processing industry. Interactive FAX voice processing is a burgeoning sub-technology area and has required specialized technical advancements to provide efficient voice-activated FAX systems. The technical advancement required to make FAX voice processing and other advanced voice processing systems feasible have not heretofore been adequately developed. There is a long-felt need in the art for a general-purpose system which can effectively, economically, and efficiently provide these technological advancements and which will integrate the above-mentioned other voice-based technologies in the voice processing field.

Examples of such systems for data reception and projection over telephone lines are disclosed in U.S. Pat. Nos. 4,481,574, DeFino et al., and 4,489,438, Hughes. Both the DeFino et al. and Hughes patent teach hard-wired systems which interface to telephone lines and computers to provide telecommunications applications. However, the systems disclosed in the DeFino and Hughes et al. patents generally perform the telecommunications transactions in hardware, thus requiring expensive and bulk equipment to accomplish these applications.

All of the above-referenced patents disclose voice-based systems which are discrete and which perform narrow, limited voice-based transactions. If a customer needs a voice messaging system, a device such as that disclosed, for example, in the Matthews et al. patent could be purchased. However, if the customer also needs a system to interact with callers and to answer routine questions about the status of, for example, their bank accounts, a separate IVR system would be necessary. Similarly, if a customer needs to perform retail ordering and accounts management, a separate CP system such as that disclosed in the Mehta et al. patent must be purchased. Thus, it can be seen that the problem facing a customer who requires multiple voice processing functions is that of the proliferation of a multitude of special purpose systems that are expensive to purchase and to maintain, and which potentially process telephone calls in separate and disjoint manners.

An illustrative example will provide to those with skill in the art an appreciation of the magnitude of this problem. Consider a bank that allows its users to inquire about the balance of their account using an IVR system, but must now transfer a call to a VM system if the caller wishes to leave a message for an officer of the bank that could not be reached. This creates several problems for both the bank and the user. First, the bank must purchase and maintain at least two voice processing systems, an IVR system and a VM system. Second, the user must wait while one system addresses the other system to provide the new voice processing function. Third, the bank has no way of getting a consolidated report of the handling of a given call from start to finish. Fourth, if the user decides that since the bank officer is not available and the IVR system can provide additional information to answer a particular question, the transfer back to the IVR takes a considerable amount of time and is complicated since the user must usually enter the entire identification password information again, thereby leaving the bank without any way to trace a particular call as it is routed from one discrete voice processing system to the other.

As more discrete voice processing systems proliferate in a single environment, the problem of multiple disjoint systems becomes even more complex. There is a long-felt need in the art for methods and systems which integrate the various disparate voice processing functions to provide a voice processing system which effectively and economically provides all of the desired voice processing function for a customer. This need has not been fulfilled by any of the prior voice processing systems heretofore discussed, which only focus narrowly on one particular sub-technology in the complex and ever-growing array of voice processing sub-technologies.

A proposed solution to solve this long-felt need has been to connect a VM and IVR system together through a signalling link that coordinates the two systems. This link allows the systems to exchange calls with proper information relating to each call and which generates consolidated report. However, the customer must still purchase discrete systems, and this solution is akin to suggesting that

the customer purchase a personal computer with a word processing package of choice, another personal computer with, for example, a spreadsheet program, and yet another personal computer with a graphics program. Clearly, this is a cost prohibitive and ineffective way of performing a plurality of voice processing tasks and is not acceptable in light of the realities of today's business markets.

Another proposed solution to the integration problem has been to package two or more discrete systems in a large cabinet. Usually, systems having a large cabinet have nothing in common except the cabinet itself. The systems may have their own separate consoles and keyboards, or they may have an A/B switch to share a single console yet still retain their individual keyboards. In all such "cabinet" systems, there is coexistence of applications but not integration of applications. Furthermore, systems which provide coexistence of applications usually provide hard-coded software in C-language, while the rest of the application development environment consists of C-language functions and programmer documentation that can only be understood by an expert programmer, but not by a customer who may require versatility and ease of use. Thus, the aforementioned integration attempts do solve the long-felt need in the art for a truly integrated voice processing system.

Yet another attempted solution to the integration challenge has been to use a fixed VM system or a fixed IVR system and modify the resultant composite system to provide VM and IVR functions for execution in tandem on a common computer. The results of such machinations have been mixed, and the customer generally ends up with an inflexible VM or IVR system wherein the limitations and problems of one half of the system dictate the abilities and utility of the other half.

Examples of attempts at integration can be found in U.S. Pat. No. 4,792,968 to Katz. The Katz patent discloses a system of analysis selection and data processing for operation and cooperation with a public communication facility, for example a telephone system. See Katz, col. 1, lines 57-60. The systems disclosed in Katz provide methods of selecting digital data to develop records for further processing and allowing a caller to interface directly with an operator. See Katz, col. 1, lines 62-68. Another example of an attempt at integration may be found in U.S. Pat. No. 4,748,656 to Gibbs et al. The Gibbs et al. patent discloses an interface arrangement implemented on a personal computer to provide business communication services. See Gibbs et al., col. 2, lines 8-12. The personal computer interprets appropriate control signals which are then forwarded under control resident software to activate a telephone station set and provide communication services. See Gibbs et al., col. 2, lines 18-28.

Another integration attempt is disclosed in U.S. Pat. No. 4,893,335 to Fuller et al., which teaches a telephone control system that produces control signals which are programmable to provide a variety of control functions to a remote user, including for example, conferencing and transferring functions. See Fuller et al., col. 2, lines 7-44. However, in all of the above-referenced attempts at integration, only limited applications are achievable and significant problems of interfacing the different voice transactions are encountered. These aforementioned attempts at integration simply do not provide high level and effective voice transactions.

The inventor of the subject matter herein claimed and disclosed has also recognized another problem facing the task of integrating VM with IVR and other voice processing systems. Caller interfaces present a significant problem in

integration since VM systems generally have fixed, hard-coded interfaces. In an integrated environment, this restricts the versatility of the entire integrated system, since it confines the system to the limitations of the original design of the VM interface. For example, if an IVR system provides voice responses to an airline for crew scheduling, it is unlikely that an IVR system could understand an employee number, translate it to an extension, look up the caller's supervisor and automatically transfer or drop the message in the supervisor's mailbox without querying the caller. The VM interface is usually inadequate to perform such complex tasking in an economical fashion. Thus, a fixed VM system quickly dominates the more flexible IVR system when the two systems attempt to operate together and the necessary VM caller interface is introduced in a pseudo-integrated environment. Such pseudo-integration schemes to put different voice processing applications together have heretofore simply not been able to accomplish the multifarious complex voice transactions required. Prior integrated systems do not solve the long-felt need in the art for a truly universal integrated voice processing telecommunications system.

During the evolution of the voice processing industry, VM systems have not been customized to perform according to a particular customer's unique specifications. Thus, VM-type systems were developed in mostly hard-coded traditional programming languages such as the C-language or Pascal language. In contrast, IVR systems were generally more sophisticated and employed primitive customization for particular applications. The IVR systems were thus generally designed in higher level programming language known as "scripted languages." Scripted languages merely replace the C-language or Pascal knowledge requirements of the system developer with that of the Basic language.

The common problem which emerges with the use of scripted languages is a disorientation of the system developer when designing the flow of the particular application. Furthermore, most scripted languages require several dozens of pages of basic code to accomplish even a simple programming task. Even though scripted programs can be interpreted by a programmer having less expertise than that which would be required if the software programs were written in the more traditional C-language or Pascal language, it will be recognized by those with skill in the art that after even a few pages of the length scripted code have been reviewed, the entire flow of the application becomes disjoint and escapes the normal comprehension of even the most expert programmers in scripted languages.

In order to devise ways of alleviating the problems extant in scripted software voice processing systems, the concept of a state, event and action to define applications having programming methodologies in, for example, C-language or Pascal have been developed. Example of such systems are disclosed in U.S. Pat. No. 4,747,127 to Hansen et al. The Hansen et al. patent describes methods of implementing states, events, and actions to control a real-time telecommunications switching system. The method of performing voice processing transactions in the Hansen et al. patent are accomplished using a scripted base language similar to the "SHELL" programming language used by the AT&T UNIX System V operating system. See Hansen et al., col. 7, lines 15-35.

The methods and systems described in the Hansen et al. patent are strictly limited to telecommunications switches on a PBX. While the implementation of states, events and actions to perform higher level voice transactions is desirable, the systems and methods disclosed in the Hansen et al. patent do not fulfill the long-felt need in the art for integrated voice processing systems

adaptable to provide multiple functions in a single, general-purpose computer environment and for varying customized applications. Furthermore, the use of non-traditional script base high-order programming language severely limits the adaptability of systems taught in the Hansen et al. patent, and thus the systems and methods disclosed in the Hansen et al. patent cannot be manipulated to provide integrated voice processing transactions.

Furthermore, voice mail systems have proliferated in the last few years and almost every type of company, large and small, is either utilizing voice mail systems in-house or leasing voice mail boxes from service providers. Companies are making significant investments in voice mail in terms of equipment, training, and service contracts. The productivity benefit of voice mail alone has proven to be enough of a reason for adding voice mail to the office environment. Now voice mail has become an expected productivity tool in almost all segments of the business community. To provide consistent corporate communications, companies are installing voice mail systems in each of their offices, and often networking them together. These corporate-wide installations require significant investment both in terms of equipment and in user training and maintenance. Service providers of voice mail, like the regional telephone companies and voice mail service bureaus, are installing voice mail systems in their networks at considerable expense, including voice mail machines in their central offices. In order to provide ubiquitous services, this requires the installation of thousands of machines. These installations require considerable investment in equipment, training, and maintenance. And in the case of service providers it also includes advertising, promotion, and sales expenses.

Voice mail systems from different vendors have their own proprietary user interfaces and feature sets. While most of them provide a core set of voice mail features, each one differs in the way it interacts with the callers. Advanced features differ from vendor to vendor even more so than the core features. There are no adhered-to standards in the voice mail industry for caller interfaces. Each vendor provides its own version of voice mail. While there are some efforts towards industry standard voice mail, most large vendors choose to accept in principle but ignore in practice the proposed standards, as acceptance of standards would open the installed base of these vendors to competition. The VMUIF (Voice Messaging User Interface) and the AMIS (Audio Messaging Interchange Standard) propositions for the most part remain propositions that have yet to be adopted by large, established vendors of voice mail systems.

Companies implementing voice mail on a corporate-wide bases, or service-providing companies desiring to offer voice mail services, are faced with a difficult decision. In order to obtain economies of scale in training and servicing they must install a one vendor solution. Yet, in order to lower their cost and promote competition they must be able to take advantage of equipment from multiple vendors. Bell companies find themselves in the position of having spent millions of dollars on voice mail equipment from a single vendor, and being locked into that single vendor now that all the subscribers are used to that vendor's voice mail interface. They are now forced to go back to the same vendor for new equipment, upgrades and enhancements. This makes for a continuous source of revenue on inflated prices for the large, established vendor and crushes competition.

All of the above-referenced patents disclose voice mail systems that comprise single fixed caller interfaces (call flows). If a customer was to deploy a voice mail device based upon one or more of those patents from any one of the voice

mail vendors today, that customer would either have to purchase additional systems from the same vendor for years to come or risk the chaos and confusion which would result from mixing systems from multiple vendors.

Thus, a need has arisen for a system that is flexible enough to allow configuration of caller interfaces to meet specific application requirements, and to emulate operation of other systems.

SUMMARY OF THE INVENTION

In accordance with the teachings of the present invention, a voice mail processing system is provided which communicates to callers through a telecommunications line from a digital computer common-platform. Caller communications are performed through caller interfaces that are flexible and configurable to allow for operation of the voice mail processing system in accordance with specific application requirements. The system of the present invention configures a particular caller's interface according to configuration information stored in that caller's subscriber profile record. Each subscriber's profile record includes information that is used to configure that caller's interface in accordance with that caller's preference or according to specific application requirements of the system.

According to one specific embodiment of the present invention, users can edit and store caller interface configuration information in subscriber profile records. Upon caller initiation of any voice mail transaction, the system will configure that caller's interface by accessing that caller's subscriber profile record by reference to a caller identification number. Such a caller identification number can be provided automatically by the system or input by the caller.

The system of the present invention enjoys the important technical advantage of having flexible and user configurable caller interfaces. Because the system of the present invention has such flexible caller interfaces, any caller familiar with some other system's caller interface can use the present invention without having to be retrained. Such configurable caller interfaces also allow the system of the present invention to be configured to meet specific application requirements. Furthermore, the ability of the system of the present invention to emulate other voice processing systems provides users with a second source of equipment for their voice processing needs.

DRWDESC: BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of an integrated Voice processing system provided in accordance With the present invention.

FIG. 2A is a block diagram of the interaction between vectors, objects, and events provided in accordance with, the present invention.

FIG. 2B is a flow diagram of an exemplary voice processing system integrating call processing, voice massaging, and interactive voice response.

FIG. 3 is a block diagram of major processing elements found in voice processing systems in accordance with the present invention.

FIG. 4 is a state flow diagram of an exemplary play vector.

FIG. 5A is a block diagram of application logic state tables provided in accordance with the present invention.

FIG. 5B is a block diagram of a compiled next-vector stack element.

FIG. 5C is a block diagram of a compiled speech stack element.

FIG. 6 is a state flow diagram of a voice window play vector provided in accordance with the present invention.

FIG. 7 is a state flow diagram of an exemplary voice window input vector provided in accordance with the present invention.

FIG. 8 is a block diagram of an exemplary voice window schema for a voice window vector.

FIG. 9A is a block diagram of an exemplary voice window field types for the voice window schema of FIG. 8.

FIG. 9B is a block diagram of exemplary voice window enunciation types for the voice window schema of FIG. 8.

FIG. 9 is a block diagram of available exemplary voice window field attributes for the voice window attributes for the window schema of FIG. 8.

FIG. 10 is a block diagram of an exemplary compiled vector element for the voice window schema of FIG. 8.

FIG. 11 is an exemplary compiled application logic state table for a voice processing system provided in accordance with the present invention.

FIG. 12 is a block diagram of a preferred methods for creating dynamic SQL statements for voice windows.

FIG. 13A is a block diagram of an exemplary message envelope which integrates voice processing functions.

FIG. 13B is a block diagram of exemplary message segment attributes for the message envelope of FIG. 13A.

FIG. 14 shows a call flow depicting the use of vectors to achieve a flexible and modifiable voice mail system. This flow chart tracks the various vectors involved in determining the voice mail caller identification, retrieving the caller's profile record, and branching into the remaining call flow logic based upon the caller interface preference contained within the caller profile record.

DETDISC:

DETAILED DESCRIPTION OF THE INVENTION

Referring now to the drawings wherein like reference numerals refers to like elements, a telecommunications system provided in accordance with the present invention is illustrated. In preferred embodiments, the telecommunications

line 30 may in fact be one integral line, or can be separate lines as shown here. The user preferably activates the vector protocol, which is also stored on the relational database 60, through telecommunication line 30 to act on the object and produce the voice transaction event. Interface 50 translates the user command into a standard query which is recognizable by the operating system of the general-purpose computer 40 as it communicates with the relational database 60.

The relational database 60 stores the vectors, objects and events which are used to drive the telecommunications systems provided in accordance with the present invention. It will be recognized by those with skill in the art that relational databases 60 may in fact be an integral part of computer 40, or may be stored in a separate digital computer that is connected through yet another communications line to computer 40. In this fashion, many digital computers 40 may actually have access to a single relational database 60 should the particular customer application require such an arrangement. Whether the relational database 60 is an integral part of general-purpose computer 40 or a separate database at a remote location, all such arrangements and equivalents thereof are contemplated to be within the scope of the present invention.

Furthermore, interface 50 may also communicate user queries to, for example, other mainframe computers 80, or yet other outside computer based hardware 90 that is adapted to understand the queries posed through interface 50 and further provides voice transaction events through telecommunications lines 70 after an object has had its state modified by a vector provided in accordance with the present invention.

In yet further preferred embodiments of telecommunications systems herein described, the user receives voice transaction events at a handset 100 which is actually part of the TOUCHTONE telephone 20. The voice transaction events which are output to the user, shown generally at 110 may, for example, prompt the user to activate yet another vector protocol in a telecommunications system stored on the database 60 or other computer equipment, or prompt the user to end his or her communications with the system.

One of the main features and advantages of the present invention resides in the fact that the application development environment and telecommunication system itself is realized in a general-use computer common-platform with industry standard architecture. This includes, an industry standard communications protocol. In further preferred embodiments, the general-purpose computer is an American Telephone & Telegraph (AT&T) 386-based personal computer utilizing the UNIX operating system. In still further preferred embodiments, interface 50 comprises the "standard query language" (SQL) interface which is an International Business Machine (IBM) standardized interface that is recognizable by a large number of databases and computer systems commonly available in the industry today.

The SQL interface converts the user's commands over the telecommunication line to standard queries which can be communicated to the relational database 60, mainframe computer 80, or other computer hardware equipment 90 having vectors, objects and events stored therein. In still further preferred embodiments, the relational database 60 provided in accordance with the present invention is also an industry standard database which communicates with general-purpose computer 40 through the SQL interface. The inventor of the subject matter herein claimed and disclosed has determined that the IBM SQL

standard based INFORMIX database is a preferable database which can be accessed by the user and which stores the vector protocols, objects, and events used by the voice processing systems provided in accordance with the present invention. While other standard databases and interfaces may be utilized in voice processing systems having state vector architecture as described herein, the industry standard SQL interface and INFORMIX database have been determined to provide store and acquisition systems which are widely used by all facets of the computer industry. Since these standard architecture systems are preferably utilized in accordance with the present invention, telecommunications systems described herein are easily modifiable and adaptable for virtually any customer application or to any particular specification of voice processing transactions known today or which will become known and desired by customers in the future.

Through the use of the SQL interface with standard TOUCHTONE telephones, telecommunications systems provided in accordance with the present invention interface to the INFORMIX database and various other computer hardware for sophisticated applications which are required by certain customers and which cannot be integrally provided by any other voice processing systems in the present sub-technology groups. These advantageous results have not heretofore been achieved in the voice processing art and represent a significant long-term solution to the integration of voice processing functions which until now have functioned in discrete operating environments, thereby greatly increasing the costs to customer of operating with one or more of these different and discrete voice processing systems.

In accordance with the present invention, customer applications of telecommunications systems are developed by using programmed "objects," determining the state of the programmed object, performing an action on the object, determining a computer event caused by the application of the action onto the object, thereby producing a voice transaction, and disposing of the event so that the process may be repeated is so desired. The disposition of the event may cause another action on the same object or, in preferred embodiments, it may force a permanent or temporary change of reference to one or more objects.

Objects provided in accordance with the present invention are defined on a microscopic scale to perform single functions and for detailed operations. However, groups of objects so provided herein by the integrated voice processing system of this invention may also be formed to facilitate operations on a higher level. To this end, objects may be as detailed as, for example, a spoken name contained in a VM application of a person sending a voice message, or as broad as an entire mailbox containing all voice messages, facsimiles, or electronic mail for an individual.

Similarly, actions may also be microscopically defined for a high degree of detail, while at the same time they may be grouped together in order to hide unnecessary detail from the user accessing the telecommunications system. A detailed action may, for example, play a spoken name from within a VM application, and another action may play, for example, the date and time that the message was sent, while yet another action plays only a particular desired segment of the message. Detailed actions may also be combined into a group of actions which are accessed on the relational database by a single reference commanding a voice message envelope to play the entire contents in a manner determined by arrangement of the detailed actions with the group. All such application of objects, actions and events are encompassed within the scope of

the present invention.

In still further preferred embodiments, multiple objects, actions, events, and event disposition instructions may be combined together to form an "Application State Logic Table" (AST). An AST implements all functions and features of a VM system and comprises detailed individual objects and actions as well as potentially several smaller groups of objects and actions. However, in accordance with the present invention, the AST may be treated as another single object, thereby allowing flexibility of integrating multiple applications.

The actions, events, and objects described herein are adaptable and designed to process any application in the voice processing universe. In accordance with the present invention, processing a voice transaction is facilitated through the use of "vector" protocols. In reality, vectors are software programs which perform the various functions that a customer desires for a particular voice processing system. Since each telecommunication system described herein is an object-oriented, event-based system adaptable to perform any of the functions presently available in the voice processing rubric, each system is "vector customized" depending upon the particular voice processing features which a customer may desire for its system. Preferably, the vectors are programmed in C-language but application developers merely used them by reference and need not concern themselves with the details of the C-language code, thereby allowing application developers not fluent in scripted computer languages to develop applications quickly and to provide layered multiple applications as the particular needs change without having to be confined by the specifications of an earlier application.

Since the vectors are not "hard-wired" and are adaptable through changes in the C-code, telecommunications systems provided in accordance with the present invention completely encompass all types of voice processing applications. Referring to FIG. 2A, an illustration of the interaction of the vectors, objects, and events is illustrated. The vectors 120 are stored on the computer in the INFORMIX database. In order to facilitate speed and ease of operation, the AST comprising a plurality of vector references and arrangements is compiled rather than used in source code form. Many different kinds of vectors are available and can be accessed by the user. The vectors 120 thus generally comprise a number of routines such as, for example, "play a message" 130, "record a message" 140, "dual tone multifrequency input" (DTMF) 150, "call back" 160, "transfer" (XFER) 170, and any other vectors which are particularly desired to implement a particular customer demanded voice processing function.

The above-referenced vector protocols provide many different kinds of voice transactions which heretofore have only been available in separate discrete voice processing systems from the different sub-technology areas. For example, the "play a message" vector 130 and "record a message" vector 140 are generally recognizable as VM functions. However, the send host message vector 150 and the wait host response vector 180 are generally recognizable as providing IVR-type functions. Similarly, the XFER vector 170 is, for example, recognized as a CP function. Thus, by using "soft-wired," that is, software programmable vectors which are easily modifiable and adaptable in the C-language, many different kinds of voice processing functions are available on a single telecommunications system provided in accordance with the present invention. Thus, a customer need not purchase many different types of voice processing systems, and the resulting customized vector-oriented systems described herein are generally two to two-and-a-half times less expensive than hybrid, pseudo-integrated voice

processing systems which have been used before. This presents significant cost savings to the customer and evinces the great advantages inherent with the telecommunication systems provided in accordance with the present invention.

In further preferred embodiments, the vectors 120 operate on a number of system-defined objects 190. The objects 190 are preferably also stored on the INFORMIX database which may be an integral part of the general-purpose computer or may be found at a location remote from the computer. The objects may also be stored in other mainframe computers or computer hardware devices when a particular object must perform a function associated with these other remote systems. The objects have "states" associated therewith which correspond to a particular place that the object is in time, after having been operated on by a vector protocol, or which may be an existing initial state before such operation. Depending upon the extent and sophistication of the telecommunications system wherein a number of users might have simultaneous access to the system causing vectors to operate on the objects, a multitude of states may exist for the object at any one time in light of the voice transaction event produced by the object that will be output to the particular users.

The vectors 120 operate on the objects 190 to change the objects' states in response to users, queries to the system. In further preferred embodiments, the telecommunications system is adapted to keep track of all of the state operations performed on the objects in time so that the particular states of each of the objects are always available to the system. The vectors 120 change the state of the objects by performing actions 200 on the objects defined by the vectors. Vectors may ask, for example, to call a digitized voice for a VM function, an account password for an IVR voice transaction, or to call a number of software attendants in a CP system. Each time the vectors 120 act on the objects 190, the objects' states are changed, thereby producing events 210. The events 210 are output to the user in a recognizable fashion so that the user can make a decision concerning which way to proceed as it accesses the telecommunications system.

Many types of voice processing transactions or events are available to provide voice processing functions. For example, a "time-out" (T/O) event 220 holds the system in a particular mode for a predetermined amount of time. Additionally, "error" events 230 and "OK" events 240 are available to the system, events which require additional user "input" 250 may be implemented in the system, as well as other events which provide sophistication and more involved user interaction with the system. The events preferably produce user-recognizable output, or promptings to the vector protocols in an AST environment, to produce new object states and further events which may finally be output to the user. Any voice processing functions which have been implemented in the past on discrete systems are thus programmable in a telecommunications system provided in accordance with the present, and the events so produced are stored on a database to be available to a user when the user accesses a particular system.

Referring to FIG. 2B, an exemplary flow diagram of state vector operation in a telecommunications systems provided in accordance with the present invention is shown. The particular exemplary embodiment shown in FIG. 2B integrates a number of voice processing functions, wherein circles represent vectors, rectangles represent objects, and triangles represent particular events output by the system.

As the user accesses the system, a "welcome" object 260 is first reached which provides a welcoming recording heard by the user when the system picks up. A "play" vector 270 acts on the welcome object to change its state, thereby producing an "OK" event 280 which outputs the voice transaction found in the welcome message. However, if the user accidentally inputs a digital signal from the TOUCHTONE telephone, thereby forcing the play vector 270 to operate on the TOUCHTONE telephone object 290 for example, the TOUCHTONE telephone object in a new state produces an "error" event 300 which is output to the user. In such a situation, a "return" vector 310 is then accessed on the relational database to operate on the welcome object, thereby producing yet another OK event 320 and replaying the welcome message, this time perhaps with a warning to wait for a next output prompt.

In this example, the OK event 320 causes play vector 270 to change the state of the welcome object 260 which then further outputs another OK event 330. At this point, another vector 340 plays a message to the user. After the user hears the particular voice transaction message, the system then waits to determine whether a T/O event 350 occurs, that is, the user has not taken any action to access another vector. If T/O event 350 occurs, then yet another vector is accessed 360 which acts on the operator object "0" at 370, thereby connecting the user with a live operator to determine if there is an input problem or misunderstanding. Other events, vectors and objects could then be accessed further by the user at 380.

However, if the user does not allow a T/O event to occur and wishes to record a message in the system, a record vector 390 is accessed, producing an event, "1" at 400 indicating to the system that a message will be recorded by the user. The system then accesses an "end" vector 410 to output an ending signal to the user and to instruct the user to hang up the handset. Alternatively, other vectors are accessible by the user after recording a message, the other vectors 420 producing yet other event-based voice transactions which are recognizable to the user. The only voice processing limitations placed on the system of FIG. 2B are those found in the specifications for the particular system required by the customer.

Upon examining the exemplary voice processing system illustrated in FIG. 2B, it can be seen that even this simple system integrates many aspects of voice processing functions which were only traditionally available on discrete systems. Since play vectors 270 are available, the system behaves as a VM system. Since the basic functions required to implement VM are not unlike the functions required to implement IVR and CP, the same vectors are used for multiple purposes in preferred embodiments. On the other hand, since messages are played to the user according to the play vector 340 and may be recorded by the user according to the record vector 390, the system also integrates basic VM functions. Similarly, since the user can access event-based attendants or operators at 370 according to the call vector 360, CP aspects of voice processing transactions are also available. The customer who has provided the specifications for the system of FIG. 2B has therefore attained a very versatile customized system which has eliminated the need for three discrete VM, CP, and IVR voice processing systems that would be available from separate vendors. The state vector architecture operating on system-defined events thus provides flexibility, and allows for total integration of what have previously been completely separate, unrelated voice processing tasks.

Referring now to FIG. 3, a more detailed block diagram of the main processing elements of integrated voice processing systems provided in accordance with a preferred embodiment of the present invention is shown. Preferably, communication logic state tables 430 are generated by the system so that processes found and generated by data communication servers 440 can perform run-time operating instructions for the system. The communication servers 440 are responsible for communication with external computers and other systems which interface with voice processing systems described herein. Yet another server 450, which is an alarm/log server, is interfaced with the data communication servers 440 to provide alarm and other type functions for the system.

The heart of the voice processing system is the run-time executive (RTX) block 460. RTX 460 in a preferred embodiment provides the main control processes for voice processing systems provided in accordance with the present invention during operation. RTX 460 is an object-oriented, run-time programmable finite state machine. The system alarm/log server 450 handles messages from all other servers in the system and logs and stores the messages. Additionally, alarm/log server 450 controls a triggering of system alarms, including audible, visible, and hard contact closures for remote sensors. In further preferred embodiments alarm/log 450 also manages the logging of system activity and usage data provided by the RTX processes. The alarms are stored in an alarm log 500 which is a data table.

The RTX preferably executes instructions which are stored and contained in an application logic state tables (AST) block 470. The application logic state tables found in AST 470 in preferred embodiments are generated by an application editor 480 (APE) which is preferably a 4GL application editor. Furthermore, application logic state tables are also generated by a GUI application editor (APEX) shown at block 490.

The APE 480 allows application developers to arrange vectors provided in accordance with the present invention to formulate the applications desired by the customer. Similarly, APEX 490 is preferably a formatted, screen interactive development program that uses a graphical interface instead of using line graphics and text-based interfaces. APE 480 and APEX 490 thus generate the communication logic state tables stored in block 430. The state tables for voice processing systems provided herein which are stored in block 430 and AST block 470 are further coordinated by APE 480 and APEX 490 to facilitate asynchronous processing of the application logic.

In a further preferred embodiment, the off loading of communications to the data communication servers 440 allows RTX 460 to issue requests and receive responses without blocking data flow. Additionally, because data communication servers 440 and RTX 460 are interactive, it is not necessary for RTX 460 to be concerned with the particular details of protocols and external computer interfaces to the system. This provides the advantageous result that the processing of the voice transactions is divided into two sets of processes, thereby resulting in inherent support for network-based processing.

Preferably also, the data communications servers 440 are driven by object-oriented application logic found in RTX 460. The SQL servers 510 function very similarly to data communications servers 440 and are responsible for front-ending relational databases, and enabling the RTX 460 to interact with relational databases using SQL statements. Thus, RTX 460 dynamically generates

SQL statements during run-time to store and retrieve information to and from relational database 530 which can be considered as the system database.

The SQL processor 510 executes the SQL statements on behalf of the RTX 460 which allows RTX 460 and SQL server 510 to operate asynchronously. This further provides the advantageous operating result of a division of processing between the two separate sets of processes which promotes network-based processing in a true front end/back end, client/server processing environment.

The exchange of information between RTX 460 and SQL server 510 or alternatively between RTX 460 and data communication server 440 is preferably conducted in well-defined boundaries across individual domains. This localizes processing and reduces the flow of unnecessary data which is encountered in prior storage servers of other network-based operating systems. An outcall server 540 is responsible for generating "outcall lists." Outcall server 540 is driven by SQL statements and selects telephone numbers for outcalls. The SQL statements for the outcall server 540 are preferably entered by a human operator or alternatively, they may be generated by the RTX processor 460. In further preferred embodiments, the output of the outcall server 540 is channelled to the RTX 460 which is then responsible for placing the outcalls. The outcall server 540 may supply only limited information about each outcall to RTX 460, or it may supply detailed information. In either case, the RTX 460 may request further information from the same data base that outcall server 540 used to made the selection in the first place by communicating through the SQL server 510.

The exchange of information between RTX 460 and the various servers available to the system is preferably organized by the RTX in internal storage using a concept called "voice windows." Information is exchanged between the voice windows stored on RTX 460 and its associated servers using formatted buffer exchanges over UNIX communications facilities.

A broadcast server 550 provides mail sorting and distribution for the integrated voice process systems provided in accordance with the present invention. Broadcast server 550 again preferably uses SQL statements to retrieve information from the relational database 530, thereby making sorting and distributing decisions. In preferred embodiments, broadcast server 550 sorts, collates, deposits, and broadcasts mail, including voice messages 560, facsimiles 570, and electronic mail 580, as well as other voice processing applications which are desired by the customer by its particular voice processing system as described herein. Broadcast server 550 also provides network transmits and distribution of inbound network mail. The broadcast server thus periodically cycles through pending requests and periodically looks for new requests. In addition, RTX 460 may invoke broadcast server 550 for "time special" or urgent deliveries.

RTX 460 is a finite state machine and it is dynamically programmed at run-time with instructions from AST 470. These application logic state tables are generated by either APE 480 or APEX 490 which allow application developers to specify the placement of objects, actions to be taken on the objects, and the handling of various events that result from the specified action.

Referring now to FIG. 4, there is illustrated an exemplary vector which encompasses an object (in preferred embodiments a speech phrase), object states, an action, and a disposition of various events. The vector may have several states, events, and disposition of events internal to it; however, there is in

general provided only one entry and exit point for the vector. The vector enters at 610 where the circles in FIG. 4 represent events such as "wait digits" 620, validation 630, T/O limit 640, and "done" where the vector exits at 650. The action of this particular vector is to play a speech phrase. Various disposition events are available such as an OK event 660, disconnect event 670, a "play done" event 680, a "digit input received" event 690, a T/O exceeded event 700, a disconnect event 710, another "error exceeded event" 720, a digit validation event 730, and yet another OK event 740. Thus, the play vector begins at 610 with an entry played so that the system waits for digits at 620, checks for digit validation at 630, or a T/O limit at 640, potentially checks for error limits at 600 or T/O limits at 640, potentially checks for error limits at 600 or T/O limits at 640, and exits at 650 following a disconnect 670. Upon exiting at 650, the vector calling mechanism checks the exit status of the vector and consults a vector's "next-vector stack" for the dispatching of the next vector.

State Table Architecture

With reference to FIGS. 5A, 5B, and 5C, the relationship of the various logic state tables that constitute the vector of FIG. 4 are shown. In the vector table of FIG. 5A, a vector 750 is the current vector that has been dispatched by the voice processing system. Associated with the vector table is an extended vector table having a vector 760 which provides "attributes" for the vector 750. The vector's attributes are the particular codes which direct the vector to perform state modification of an object as described herein. In the next-vector stack table, a next-vector stack 770 is accessed by the vector 750 to point to the next-vector available in the stack. Similarly in the speech stack, a next speech stack 780 is accessed by vector 750.

Vector 750 accesses an extended vector 760 through a reference 790 which references through the top of the extended vector stack. Similarly in the next-vector stack, the vector 750 references the next vector 770 through a reference 800 to the top of the next-vector stack. Speech associated with the vector is referenced through 810 on the speech stack. Reference 810 references through the top of the speech stack elements.

SQL statements associated with the vector are referenced through a reference 830 to an extended parameter stack table to a specified entry 820 within the table. The vector 750 comprises codes which tell the system, for example, which extended vector to use, which extended parameter is next, which vector is next in the stack, and which speech element is next in the stack. A plurality of vectors so described having the elements 750 through 830 are stored in the AST 470 as a set of logic state tables.

FIG. 5B shows the compiled next-vector stack element in an exemplary embodiment. The next-vector stack element may comprise a "key" field 840 which points the vector to the next event available in the system. The next event shown in this exemplary embodiment is a "macro call" in field 850 which tells the vector to call and search for the next vector stored in field 860. In a similar fashion, a compiled speech stack element shown in FIG. 5C comprises for example, an archive field 870 which tells the vector where to go in a database to retrieve a speech element, according to a particular format in a format field 880. An offset field 890 holds a code to tell the vector at which point in the archive to retrieve the speech, and a length field 900 tells the vector the particular length of the particular speech element which will be retrieved.

Voice Window Architecture

As discussed above, in preferred embodiments the exchange of information between RTX 460 and the data communications server 440, the SQL server 510, and outcall server 540 is organized using voice windows provided in accordance with the present invention. Referring to FIG. 6, there is shown an exemplary voice window protocol organized to manage the information exchange between the processes heretofore mentioned, in between the particular voice transactions and uses of the system. In the exemplary embodiment of FIG. 6, the state transitions within the voice window vector for an output or protected field are illustrated. The voice window structure of elements plays data at 910 and begins with a play label 920 that acts as the entry point for the vector. Similarly to the play vector shown in FIG. 4, the vector shown in FIG. 6 has an object which in this case is the voice window field, and has several states such as the "play data" state 910, "play a label" state 920, the "next field" state 930, and a "done" state 940, which is the exit. The vector of FIG. 6 plays the voice window field, and contains a disposition of various events, for example, event 950 which is "play done," event 960 which is "digit received," event 970 which is "field found," event 980 which is "disconnect," event 990 which is "done play data," event 1000 which is "digit," event 1010 which is another "disconnect," and event 1020 which is "no next field."

Upon entry at play label 920, the vector executes the first field in the voice window and continues execution through the entire set of voice window fields, unless prematurely interrupted by a terminating event. Using this vector, an application dispenses information to the caller one information field at a time. On entry at play label 920, the vector preferably plays the field label associated with the first field, and either plays to completion and leaves the play label state through event 950 play done, encounters a caller input event 960, or gets a disconnect signal from a telephone switch at disconnect event 980. If either done event 950 or digit event 960 are encountered, the state preferably transitions to state 910 play data, and the system commences playback of the data contained within the voice window field.

Play data state 910 is terminated by play done event 990, digits received event 1000, or disconnect event 1010. In the case when done event 990 or digit received event 1000 is encountered, the program then dispatches the next field state 930 which causes the program to bump the field pointer to the next voice window field available in the vector.

If next field state 930 is able to bump the field pointer, the cycle starts over with found event 970 and transitions to play label state 920, otherwise the vector is prepared for exit through the done event 1020 and the done state 940. In cases where the program is interacting to one of its states through the telecommunication channel, a disconnect signal from the telephone switch that is connected to the telephone channel will cause immediate exit of the vector through done state 940. Upon exit from this vector, the vector calling mechanism checks the exit status of the vector and dispatches the appropriate vector based upon the entries and the returning vector's particular specified next-vector stack. This is identical to the return to post exist processing of the play vector exit through 650 as discussed in reference to FIG. 4.

State Transitions

To better understand the voice window state transitions extant in FIG. 6, FIG. 7 illustrates the allowed voice window state transitions. When executing an input allowed field, the vector entry point at 1040 is a "play label" state. The vector steps through all the voice window fields, plays the voice field label

through play label state 1040, and if a "digit input" is received from the caller at digit even 1130, the next state dispatched will be "get digit" state 1030, which gets the digits and waits for the number of digits expected. If all the digits are received in time, a "digit" event at 1160 causes dispatch to the next state to the "populate" state 1070 which populates the voice window field with the digits. The populated event 1070 then returns and the next field state 1080 which bumps the field pointer to the next voice window field after the done state is reached through a "none" event 1170 signifying that no text field is available. If "next field" state 1080 finds another field in the voice window, then the cycle starts over with the new field through "next field" event 1110 to the play label state 1040. However, if a T/O event 1150 takes place after the "get digit" state 1030 occurs, then time out event 1150 dispatches the "T/O limit" state 1050 which checks for the number of time outs that have taken place so far, and either causes an "OK" event 1090 to dispatch the play label state 1040, or causes the exceeded event 1140 to dispatch the done state 1060.

The combination of vectors in FIGS. 6 and 7 can be used to create a "super-vector" which handles both output or protected fields similar to the play vector shown in FIG. 4, in preferred embodiments. This super-vector may then be deployed to handle the exchange of information between the system processes and voice transaction events to the users. Using the super-vector on voice window fields, the caller and the system programs are able to exchange information with a mixture of both protected and unprotected voice window fields.

Refinements and enhancements may be added to the vectors illustrated in FIGS. 4, 6 and 7. For example, information validation prior to population of the voice window field at population state 1070, verifying the data input to ensure that direct information is entered by the caller, reentry of a date, opportunity to accept partial input which is used when the program is able to narrow down from the partial input when a caller is attempting to offer appropriate help, and others are all potentially addable to the system. Additionally, voice programs may be used to build program/caller interfaces, which are interfaces used to exchange information between clients and the servers 440 and 510. For example, RTX 460 may request SQL server 510 to reply with the results of an operation and any information retrieved from the relational database 530. As part of this request from RTX 460 to SQL server 510, RTX 460 attaches a reference to one of its voice windows which receives the results of the database operation. When SQL 510 completes processing of the request, it responds back to RTX 460 using the reply to address contained in the original request. Upon receiving of a response, RTX 460 reloads the voice window reference and populates the window with information contained in the response packet. In this fashion, voice windows provided in accordance with the present invention are used to exchange information between users and the user interface program, and between user interface programs and relational databases.

Thus, the use of voice windows along with the creation of SQL statements from voice windows provides an extremely powerful and flexible method of facilitating user interfacing to relational database through SQL statements. The cooperation between RTX 460 and SQL 510 may also be extended in accordance with this invention, to cooperative processing between RTX 460 and the data communication servers 440. In this embodiment, data communication server 440 perform server functions by interacting with an external computer directly using SQL statements, wherein the external computer is in an environment which is foreign to the integrated voice processing system. For example, data communications server 440 may be a terminal emulation program using IBM 3270, Unisys

Poll/Select, or UNISCOPE with Unisys terminal emulation, IBM 5250 terminal emulation, digital VT 100 terminal emulation, IBM LU6.2 peer-to-peer communications, or any other variety of communications protocols. Further in the case of the LU6.2 pair-to-pair protocol, the peer process on the foreign host external computer may also be an SQL server acting on an SQL front-ended database residing on the foreign machine.

Vector State Logic

While the data communication server 440 is custom designed for any type of communications protocol, the communication logic state tables 430 contain the vectors which define the "personality" of the data communication server in each customer application. Thus, as with the RTX 460 assuming multiple personalities based upon the arrangement of vectors in the application logic state tables 470, a single IBM 3270 server program in preferred embodiments serves multitudes of applications based upon the arrangements of the vectors in the logic state tables. By using vectors and windows with dynamic SQL statements, the flexibility inherent in creating powerful interfaces between the users and the computers in a myriad of applications is multiplied. The interface so created is not limited merely to users communicating with computers, but could also be used to communicate between two computers in a voice processing system, wherein one of the computers emulates a user. The use of this kind of computer to computer interface is especially adaptable to customer specification requiring stress testing or benchmarking systems applications.

Voice Window State Logic

Voice windows can be viewed as collections of related voice fields, in a manner similar to a display in graphical window systems which are collection of display fields on display terminals. Each of the voice fields has particular characteristics and attributes, and an application developer can define a voice window in accordance with the present invention similar to defining the layout of a display screen in a display window system. Once a voice window is so defined, it is then referred to by vectors as objects to operate on. Each voice window will then have at least one voice field and a name indicating the ownership of the field by a specific voice window.

Referring to FIG. 8, a voice window schema is illustrated herein each voice window has a name in a window name field 1190. The field name 1200 identifies the field within a window wherein window name 1190 and field name 1200 together represent a unique, fully qualified name for the voice window. This combination is used to reference the field throughout the application logic state tables 470. Label 1205 defines a field type in the voice window and label 1210 delineates the input size which preferably is a number of digits that are solicited for input from a caller. Label 1220 contains the voice label for the particular field which is much like the title or a heading for the information displayed on the display terminals in a window graphics system. Voice label 1220 is further preferably played by the vector window before either the data contents are played, or an input is solicited. The cross window name 1230 contains the name of the voice window that is used to resolve the window type by referencing to a cross window which is simply a window which works in tandem with the present voice window to accomplish a voice transaction. Cross-field name 1240 contains the name of the field for resolution of the cross field reference as described above. Validation function 1250 contains the name of the optional user written, C-language function that may be called upon for data validation or transformation. Validation functions in preferred embodiments are bound at run-time and need not be bound together at link time to the RTX

run-time executive.

A conversion format 1260 is used to perform transformation of data while populating the field in the case of unprotected fields and announced the field contents on protected fields. An enunciation type 1270 preferably defines the form of enunciation to be used for playing data from a field. Input delimiters 1280 provide a list of digits that will signal an end of the input from the caller whenever any one of the digits is encountered. Finally, field attributes 1290 modify the program behavior of the vector executing the field with a further combination of attributes. In the case of window field types 1205, window enunciation types 1270, and window field attributes 1290, various possible values for each of these elements are possible. Referring to FIG. 9A, voice window types such as a character 1300, a filler 1310, or a simple time stamp 1320 are possible window field types. In the case of a unique file name 1330, a particular data file type that causes the field to be populated by a string of characters representing a file name that is guaranteed to be unique across the entire system is specified. This particular file name may be used to deposit various forms of messages for recipients. It may then be used without modification for broadcast to other recipients within the same system or the network without the possibility of file name collisions.

A "constant" field type 1350 identifies a constant within the voice window to be applied to the vector transformation. The voice window field type "reference" 1360 indicates that the field contains no data and that all attributes and characteristics are to be applied to data residing in another field either within the same voice window or across another window. The "copy" voice window field type 1370 preferably states that this field is to be populated with data from another field, either from within the same window or across another window. The "sequence" voice window field type 1380 causes the contents of this particular field to be incremented each time this window is executed.

Referring to FIG. 9B, the enunciation types available to this particular voice window scheme are illustrated. The enunciation type "date" 1390, "money" 1400, "numeric" 1410, and "paired numeric" 1420 provides these particular enunciations to the voice window and are common voice processing functions. Enunciation type phrase 1430 is used when certain data are necessary as a key to locate a prerecorded voice "phrase" to be played to the caller instead of the data itself. The enunciation type "time" 1440 is populated from a C-language formatted time function all to play the time to the caller. Finally, the enunciation types "percent" 1450 and "none" 1460 provide these particular self-explanatory enunciations to the voice window for play to the caller.

Referring to FIG. 9C, attributes which modify the program behavior of the vector executing the field with a combination of attributes are shown. The "verify" attribute 1480 causes the field input to be solicited twice and compared before populating the field. "Partial input" attribute 1490 allows practical input of data to be accepted into the field, which facilitates the program to make an educated guess at what the caller is trying to accomplish if and when the caller is having difficulty using the system. The "continue on error" attribute 1500 disregards errors caused by the input or output of the field data with a resulting premature interruption or execution of the voice window by the executing vector. Finally, the field attribute "non-volatile" 1450 holds the system in a non-volatile state when certain input is received from the user requiring particular wait times.

The detailed structure of compiled vector elements as they exist on the application logic state tables is illustrated in FIG. 10. The vector name segment 1520 identifies the vector and the vector type segment 1530 defines the action that the vector performs on the objects that it operates on. A "sub-type" segment 1540 further defines the action that the vector performs when yet more complicated actions are required. A "parameter" segment 550 contains information that is interpreted by the vector based upon the type 1530 and sub-type 1540. In preferred embodiments, whenever information in a parameter 1550 is used by a vector, that information becomes the object of the vector. The parameter 1550 contains any one of several parameters, including for example, voice window names, voice field names, phone numbers, dates, times, call durations and other particular parameters defined by the particular vector in use.

In preferred embodiments, any time a value is specified as a parameter to a vector in parameter 1550, the value itself may be substituted for by a voice field name which creates a dynamic object passing mechanism that is resolved at run-time based upon information contained in the voice windows. This type of parameter information may come from the caller, from the relational database, from an external computer, or it may be a combination of all of these sources. Because of the flexibility of parameters 1550, the dynamic run-time modification of the execution logic in accordance with this invention is maximized.

An extended parameter segment 1560 contains the skeleton of the SQL statement which is used to create dynamic SQL statements using information from the voice windows. A next-vector stack segment 1580 preferably contains a plurality of next-vector dispatching instructions which are matched up against vector exit conditions. Similarly, the speech stack 1590 contains a plurality of speech references into the speech archives. These particular speech references are used to locate speech fragments to play to the caller when particular vectors call for the playback of speech on demand.

Referring to FIG. 11, an illustration of the structure of the application logic state tables is shown. A compiler version and time stamp 1600 ensures that the APE 480 and APEX 490 mark the state table so that a compatible version of RTX 460 is user to execute the application logic. A compiled channel descriptor 1610 contains information about the various telephone channels that will be serviced by the voice processing system. General information about the type of channel and its signalling characteristics are maintained in the compiled channel descriptor segment. Similarly, speech archive descriptor 1620 contains information about the particular speech segments that are used by the system.

An input/output (I/O) translation table 1630 holds an index for locating speech phrases based on data contained on voice fields if called for by the phrase enunciation 1430, which is also used to translate input into window fields. Using input translation, caller inputs may be located in the index and substituted with the contents of a translation entry whose key matches the caller input. To avoid collisions with data in a voice window which is the same as data in another voice window but applied differently, the conversion format 1260 is applied to the data before performing look-up in the I/O translation 1630.

The plurality of vectors segment 1640 contains the compiled vectors, and the plurality of voice window segments 1640 preferably contains a plurality of voice windows. Voice windows thus appear in window fields consecutively stacked together from the first field to the last field. Each window is referenced by

a window header which preferably contains information about the size and data area required to hold the field contents of the window. Windows are preferably read from a plurality of voice window segments 1650 dynamically created and destroyed in the main memory of the vector executing program at run-time. When the particular caller terminates the call, all the windows are destroyed in the main memory and the data area which has been reserved for the windows is preferably released. However, some window fields may be marked "non-volatile" by the field attribute 1510 and these windows are then used to maintain information between calls, thereby serving as a parameter passing mechanism between particular calls. This allows the system to behave in a dynamic fashion transferring data from caller to caller in preferred embodiments.

SQL Statement Creation

It is generally desired to create dynamic SQL statements from voice windows as described above. Referring to FIG. 12, a flow chart for transformation of SQL statements at run-time to executable statements from data in voice windows is illustrated. A dynamic program searches for a token at 1660 which are SQL components used a building blocks for SQL statements. If all the tokens have ended at 1680, there is a normal exit from the program at 1670. Otherwise, the token is written to the window field at 1690. If there is not a token in the window field at 1690, then the token is written at 1700 and the next token is accessed at 1660. Otherwise, at step 1720 it is determined whether the window is indeed active with a present token. If not, then the method defaults to 1710 with an error and the program is exited. If default does not occur, data is written at 1730 in a field and the method returns to 1660 for the fetching of the next token. In this matter, the SQL statements are transformed at run-time into executable statements in the voice window architecture.

Message Envelope in a VM System

With voice processing systems provided in accordance with the present invention, the main object of the manipulation of data by vectors is accomplished in integration of IVR, VM, facsimile, electronic mail (E-mail), CP, and other types of voice processing functions. A "message envelope" is created to provide integration as is illustrated in FIG. 13A. At block 1740, a "magic number" which identifies a particular file as a message envelope and the version of the program that created the envelope for backward compatibility is accessed. A segment count 1750 contains a count of message segments within the envelope, each envelope containing at least one segment. However, the envelope may contain several other segments of the same media type, or it may contain a mixture of voice, FAX, E-mail, or other data. Multiple segments of voice processing functions are used to keep a voice message intact while it is being forwarded from one mailbox to another with an adaptation from each mailbox owner. Additionally, multiple segments are used when a FAX or an E-mail message is sent or forwarded to another mailbox owner. In this fashion, multiple media message formats are contained within a single envelope.

A segment size block 1760 contains the size of the segment, while a sender identification (ID) block 1770 provides an identification of the sender of the message in the segment. If the sender ID is not known, this field will contain a zero for telephone calls from the outside where the caller ID is not available. The time sent block 1780 preferably contains the time when the message was created. Message attributes 1790 contain a combination of attributes for the various messages. Message format 1800 indicates the format of the message which is contained in a contents segment at block 1810. This format specification includes media and data compression indicators. The content segment 1810 is

preferably used by the RTX 460 to determine the medium and type of delivery mechanism to be used to deliver the contents of the content segment 1810 to the recipient. The message envelope is repeated at 1820 for the particular segment size IDs, times sent, etc., while the magic number and segment count are kept constant at 1830.

Referring to FIG. 13B, the particular available message segment attributes 1790 are illustrated in this exemplary embodiment. A "notify on delivery" attribute 1840 is set by the RTC 460 to request a subsequent session of the RTX to notify the sender on delivery of the message. A "message purge" process provided by this particular message envelope periodically purges messages that exceed a predetermined storage retention time allocated for each type of message. The purge process also informs the broadcast server if it has purged a message that should have been purged based on age, but before it could be picked up by the recipient.

A "reply allowed" attribute 1850 allows the recipient to reply to a message when sent. When the "reply allowed" attribute 1850 is set to zero, it indicates that a mail box owner who sent a general broadcast message does not wish to receive a multitude of replies in response to the general broadcast with a large distribution list. The sender of the general broadcast in preferred embodiments thus has the option of setting the reply allowed attribute to zero. A "private message attribute" 1860 disables the forwarding of a message past the recipient, and an "urgent" attribute 1870 marks the message as urgent and ensures that messages so marked are presented to the recipient before any other messages are presented.

It will be recognized that several types of vectors may be used to manipulate the envelopes and messages as described above. These include, but are not limited to, vectors to create, append, forward, save, purge, pick up, reply, broadcast, and vectors which provide other actions which are particularly used to manipulate mailboxes, envelopes, and other messages. Since the arrangement of vectors to manipulate the envelopes and messages are dynamically programmable by APE 480 and APEX 490 into AST 470, the entire personality of an integrated voice processing system provided in accordance with the present inventions is fully configurable and changeable. Thus, a voice processing system so described may be designed to emulate any existing menu structure, or a completely new menu structure for a system particularly defined.

In addition to the vector manipulation of envelopes and messages, RTX 460 preferably reads information into an out-of-message envelope using voice windows as described herein. In order for a caller to receive a message, a first vector preferably opens the envelope, and a second vector reads information from a message envelope into a voice window. The information in the voice window is then available for processing by all other vectors that are used to develop voice processing applications. Envelopes and messages may also be created using windows to collect information from the caller, or the data communication server 440 and SQL server 510. Voice processing systems provided in accordance with the present invention simply convert voice messaging into an interactive voice response application with the use of a message envelope as an object that is manipulated by vectors wherein the information contained in the envelope and the message are available to all other vectors as the information is obtained from, or bussed to, the data communications server 440 or SQL server 510. In accordance with the present invention, the vectors themselves need not be concerned with the details of the particular VM, facsimile, E-mail, or other

voice processing function, but provide a structure for accomplishment of these functions and the relationship between these functions which are stored on a relational database or other system.

Integration of Voice Transaction Vectors

In the case of a collection of vectors which implements voice mail, this set of vectors may be integrated in the same application logic state table with a set of vectors that implements other database or external computer based transaction processes using the data communications server 440 or the SQL server 510. In this fashion, several sets of vectors may be preferably combined to form a tightly integrated, multiple application voice processing system providing efficient interfaces. Examples of these applications include a sophisticated call director, a VM system, an IVR system, a facsimile store forward server, and an E-mail server.

Additionally, in preferred embodiments, groups of vectors defining various applications may be bound together in a single application logic state table 470 and a single communication logic state table 430. Alternatively, they may be separated over several communication logic state tables and application logic state tables. The separate logic state tables may then be treated as objects by a vector from within the logic table that is executable, and operates upon a named logic table external to itself. This further permits creation of a library of vectors that is reusable across diverse sets of customers. Since all these applications are under the control of the same RTX 460 and communication logic state tables 430 and application logic state tables 470, information may be freely exchanged between the various applications.

Thus in accordance with voice processing systems described by this invention, a caller may now request information about an account, listen to this information, request a facsimile copy sent to a particular FAX machine, and leave a message for its account representative without ever leaving the system. In addition, the message sent to the account representative takes advantage of the fact that the caller's account record located on a mainframe using the data communications server 440 contains the identification of the account representative and thus, the caller is never asked to identify the representative nor is the message ever deposited in a general mailbox for manual resorting and redirection. The message also now contains a customer identification, a summary of the caller's account status at the time the message was created, and identifies the fact that the caller requested a facsimile copy of the account information.

Should the representative wish to leave a message for the caller, the customer account information is retrieved into a window, and the "create a mailbox" vector is called to create an on-demand mailbox for the customer if one does not exist so that the message is then deposited into an envelope in a mailbox. The on-demand mailbox is then treated as a regular mailbox until it is aged and purged from the system. On the next call, the caller will then be informed of the waiting message and given an option to be forced to pick up the message before being given any other information.

Thus, it can be seen that a multitude of options and possibilities are opened up by the integration of voice mail, facsimile, and E-mail, with interactive voice response functions using vectors, voice windows, and SQL processing provided in accordance with the present invention and described by this exemplary embodiment. The personality of the voice processing system so

described is entirely dependent upon an arrangement which forces the callers to clear their mailboxes before being given further information about their accounts. Other arrangements could, for example, simply inform the callers of the waiting messages, while other examples would require the callers to make a specific selection to check if they have any messages. An endless combination of vectors is therefore possible, and an endless number of applications of seamless integration is provided in accordance with the present invention. Such results have not heretofore been achieved in the art and satisfy a long-felt need for integration of voice processing functions.

Configurable Caller Interfaces

Furthermore, through the use of the state vector logic described herein, voice mail systems are completely configurable to any desired interface, and can be adapted to perform all the various functions in a manner identical to existing systems the customer may already have deployed in the network. This results in systems that may be customized to fit specific requirements, and introduces a friendly voice mail system tailored to communicate with the outside world in a manner intended by the customer instead of being forced into it by the voice mail vendor. Another advantage derived due to the implementation of state vector logic in voice mail systems as described in the present invention is that a voice mail system created using the vector logic may be customized to emulate the caller interface of existing voice mail systems. This feature of voice mail systems created in accordance with the present invention will provide customers who have significant investments in their existing voice mail systems a second source for voice mail systems instead of being held captive by a single voice mail vendor. This will increase competition and lower prices. In addition the vector state logic may be used to implement multiple caller interfaces and still remain easily modifiable. Voice mail subscribers may specify their preferred caller interface at the time of subscription and be able to change it at anytime thereafter, eliminating the need for retraining themselves after moving to a location most probably being served by a voice mail system from a different vendor. These advantageous results have not heretofore been achieved in the voice mail art and represent a significant long-term solution to the closed, black-box voice mail systems which until now have functioned in a rigid manner, and have been available only from the source that originally sold the customer the original systems, thereby greatly increasing the costs to the customers often associated with single sourcing of products.

In accordance with the present invention, customer voice mail systems are developed using programmed "objects", determining the states of the programmed objects, performing actions on the objects, determining computer events caused by application of the actions to the objects, thereby by producing voice transactions, and disposing of the events so that the process may be repeated if so desired. The disposition of events may cause other actions on the same objects or, in preferred embodiments, it may force a permanent or temporary change of reference to one or more objects. Objects provided in accordance with the present invention are defined on a microspace scale to perform single functions and for detailed operations. However, groups of objects provided here by the voice mail system of this invention may also be formed to facilitate operations on a higher level. To this end, objects may be as detailed as, for example, a spoken name contained in a voice message of a person sending the message, or as broad as the entire mailbox containing all voice messages, facsimiles, or electronic mail for a subscriber.

Similarly, actions may also be microscopically defined for a high degree of detail, while at the same time they may grouped together in order to hide unnecessary detail. A detailed action may for example cause the play action to take place on an object like the spoken name of the sender, and another action may be to play the date and time the message was sent, while yet another action plays only a particular segment of the message. Actions like save and delete may be caused upon a voice message, also actions like next, first, last, and skip may be caused upon a voice message object to change the referenced object. Detailed actions may also be combined into a single reference commanding a voice message to play the entire contents in a manner determined by the arrangement of the detailed actions within the group. All such applications of objects, actions and events are encompassed within the scope of the present invention.

In still further preferred embodiment, multiple objects, actions, events, and event dispositions may be combined together to form an "Application State Logic" (ASL). An ASL implements all functions and features of a voice mail system and comprises detailed objects and actions as well as potentially several smaller groups of objects and actions. However in accordance with the present invention, the ASL may be treated as another single object, thereby allowing flexibility of integrating multiple applications. In accordance with the present invention, processing a voice transaction is facilitated through the use of "vector" protocols. In reality, vectors are software routines which perform the various functions that are required of a voice mail system. Since each voice mail system described herein is an object-oriented, event-based system adaptable to perform any of the functions presently available in the voice mail rubric, each system is "vector customized" depending upon the particular voice mail features which a customer may desire for the system. Preferably the vectors are C-language functions but application developers merely use them by reference in a non-programming environment they need not concern themselves with the details of the C-language code, thereby allowing application developers not fluent in programming languages or scripting computer languages to develop and customize voice mail systems quickly.

As discussed above, APE 480 and APEX 490 allow users (application developers) to arrange vectors to meet their specific application requirements. APE 480 and APEX 490 thus allow application developers to edit the call flow of the present invention in accordance with their particular application requirements. Appendix A to this application, incorporated herein by reference, is a representative listing of the software used to control caller interfaces to the voice mail and integrated voice processing system disclosed in this application.

As described above, the architecture of the present invention involves vector operations acting on objects to produce certain events, such as voice messages to callers. The sequence by which the vectors perform their actions is known as the call flow of the system. The software listing of Appendix A is a listing of specific vectors, those vectors, attributes, and vectors branched to once a vector has completed its action. To configure caller interfaces according to specific user applications, or to emulate other caller interfaces, a user simply edits, using APE 480 or APEX 490, the software listing of Appendix A. APE 480 and APEX 490 allow the user to change the attributes and branching sequences of all vectors. For example, play vector POgrecIn performs the action of playing a caller's voice greeting and provides the option of changing the message. As can be seen from Appendix A, a user who presses key number "5" can record a new greeting message (pressing key "5" will cause a branch to vector PORECGet). Using APE 480 or APEX 490, however, a user of the present invention may edit

play vector P0grecIn such that a new greeting message can be recorded after pressing some key other than "5", such as "1", "2", "3", or "4", for example. By performing such an editing function, users can configure their systems to meet specific application requirements, or to precisely emulate any other system's functionality. Editing of the attributes and branch sequences of various vectors allows users to configure call flows as desired.

As described above, by editing vector attributes and branches, call flows for each and every caller may be configured to meet the requirements or preferences of that caller. Consequently, a particular caller interface will be configured according to the call flow associated with that particular caller interface. Associated with every caller is a subscriber profile record, which contains that caller's interface configuration. Thus, different callers to the same voice mail or integrated voice processing system may have differently configured caller interfaces, depending on the configuration data stored in that caller subscriber profile record. When a caller calls the voice mail processing or integrated voice processing system of the present invention, the system can access that caller's particular caller interface configuration by reference to a caller identification number. That identification number can be input by the caller, or provided by serving switches or telephone networks. Each caller identification number allows access to a specific subscriber profile record, wherein interface configuration data is stored.

Configuration data generated by APE 480 or APEX 490 and stored in subscriber profile records can be presented and stored in a plurality of ways. For example, a system manager can configure all callers' interfaces and store such configuration using APE 480 or APEX 490. As another example, a system manager can create several "package" interfaces, and any caller can, by accessing the system of the present invention, choose a desired "package" interface, and have that interface configuration stored in the appropriate subscriber profile record.

Furthermore, APE 480 and APEX 490 can be used to customize the system of the present invention, by deleting, adding, or modifying vectors. Consequently, users can tailor the function of the system of the present invention to any desired operation.

Referring now to FIG. 14, a graphic example is presented by which the invention described herein accesses a particular caller's interface configuration. In FIG. 14, circles represent vectors, and triangles and lines represent events. Upon initiation of a call, play vector 1900 plays a recorded "welcome" message. If during playback of the "welcome" message, a disconnect signal is received, play vector 1900 will initiate disconnect event 1910, in turn causing hang-up vector 1920 to hang up that particular caller's communication line. After completion of the playback of the "welcome" message (event 1930), window 1940 will be accessed. Window 1940 receives a caller identification number, either directly input by the caller or from a server switch or telephone network. If no identification number is received, then disconnect event 1910 will occur and hang-up vector 1920 will hang up that caller's particular telecommunications line, or some other equivalent action, such as connection to an operator, can occur. Once a caller identification number has been received (event 1950), standard query language (SQL) command 1960 will request that caller's specific subscriber profile record. After SQL 1960 has been issued (event 1970), wait vector 1980 will be initiated to wait for the subscriber profile record to be retrieved in response to SQL 1960. If

no subscriber profile record is found (event 1990), then play vector 2000 will be initiated. Play vector 2000 will play a message that no subscriber record has been found, and then can be configured to institute several other events. For example, after playing the "no subscriber found" message (event 2010), disconnect event 1910 and hang-up vector 1920 can be initiated to hang up that user's telecommunications line. Alternatively, for example, after playing "no subscriber found" message (event 2020), another sequence of vectors and events 2030 can be invoked to subscribe that caller to the system, requesting that caller to configure his interface by inputting his interface preference, or connecting that caller to an operator, for example.

If a subscriber profile record has been found (event 2040), branch vector 2050 will configure that caller's interface. Branch 2050 will configure that caller's depending on the configuration data in that subscriber's profile record. It is understood that configuration events 2060, 2070, and 2080 are only exemplary, the present invention being able to configure caller interfaces in a plurality of different configurations. Vectors 2090, 2100 and 2110 represent the first vectors that will be called pursuant to the configuration data stored in a caller's subscriber profile record, and configured by branch vector 2050. Vectors 2090, 2100, and 2110 in turn will branch into specific call flows 2120, 2130, and 2140, once again depending on the configuration data stored in that subscriber's profile record.

While the method shown in FIG. 14 may be accomplished by a variety of software steps and in different languages and formats, Appendix A presents code operable to perform typical functions of the invention, such as those of FIG. 14. The functions of the invention, such as those depicted in FIG. 14, are performed by calls to vectors such as those in Appendix A.

There have thus been described certain preferred embodiments of voice mail and voice processing telecommunications systems provided in accordance with the present invention. While preferred embodiments have been described and disclosed, it will be recognized by those with skill in the art that modifications are within the true spirit and scope of the invention. The appended claims are intended to cover all such modifications.

CLAIMS: What is claimed is:

[*1] 1. A voice processing system for providing a plurality of voice transactions including voice messaging, call processing, and interactive voice response through telecommunications lines for a plurality of callers comprising:

a plurality of caller interfaces for communicating between said callers and said voice processing system, said caller interfaces for allowing said callers to perform said voice transactions, each of said caller interfaces comprising an arrangement of vectors, objects, and events which define attributes and branch sequences of each caller interface;

configuration means for configuring said interfaces with respective arrangements of vectors, objects and events such that said callers perform said voice transactions through said interface as configured;

storage means for storing configurations of caller interfaces; and

retrieval and communications means for retrieving said configurations from said storage means and communicating with callers through said caller interfaces configured according to said configurations.

[*2] 2. The voice processing system of claim 1 and further comprising:

said configuration means being operable to separately configure each of said caller interfaces; and

said storage means being operable to separately store each of said configurations of caller interfaces.

[*3] 3. The voice processing system of claim 1 wherein said configuration means is further operable to insert new configurations while the voice processing system is on-line.

[*4] 4. The voice processing system of claim 1 and further comprising:

means for receiving caller identification numbers; and

means for accessing said configurations of caller interfaces from storage by reference to said caller identification numbers.

[*5] 5. A method of providing a plurality of voice transactions including voice messaging, call processing, and interactive voice response through telecommunications lines for a plurality of callers to a voice processing system comprising the steps of:

communicating between said callers and said voice processing system through a plurality of caller interfaces, the caller interfaces for allowing the callers to perform the voice transactions, each of the caller interfaces comprising an arrangement of vectors, objects, and events which define attributes and branch sequences of each caller interface;

configuring said interfaces with respective arrangements of vectors, objects and events such that the callers perform the voice transactions through the interfaces as configured;

storing configurations of caller interfaces; and

retrieving said configurations and communicating with callers through said caller interfaces configured according to said configurations.

[*6] 6. The method of claim 5 and further comprising the steps of:

separately configuring each of said caller interfaces; and

separately storing each of said configurations of caller interfaces.

[*7] 7. The method of claim 5 and further comprising the step of inserting new configurations while the voice processing system is on-line.

[*8] 8. The method of claim 5 and further comprising the steps of:

receiving caller identification numbers input by callers or received from serving switches or telephone networks; and

accessing said configurations of caller interfaces from storage by reference to said caller identification numbers.

[*9] 9. A voice processing system for a plurality of callers, comprising:

a plurality of telecommunications lines for establishing connections between the voice processing system and the plurality of callers;

a digital computer operable to communicate through the telecommunications lines to the plurality of callers;

a plurality of caller interface configurations accessible by the digital computer, each caller interface configuration comprising an arrangement of vectors, objects and events which define the attributes and branch sequences of each caller interface;

selection means for allowing a caller to select a particular caller interface configuration;

configuration means operable to configure the vectors, objects, and events, and further operable to configure vectors, objects, and events while the voice processing system is on-line;

storage means for storing the caller interface configurations; and

retrieval means for retrieving each caller interface configuration by referencing a subscriber profile record accessed by the digital computer and referenced by a caller identification number.

[*10] 10. The voice processing system of claim 9 wherein the configuration means includes a formatted, screen interactive development program that uses a graphical interface providing visual manipulation of the vectors, objects, and events of the caller interface configurations.

[*11] 11. A method of voice processing for a plurality of callers comprising the steps of:

establishing telecommunications connections between the voice processing system and the plurality of callers;

selection by each caller of a particular caller interface configuration from a plurality of caller interface configurations wherein each of the caller interface configurations comprises an arrangement of vectors, objects, and events which define the attributes and branch sequences of each caller interface configuration;

arranging the vectors, objects, and events of each caller interface configuration;

storing each caller interface configuration; and

retrieving each caller interface configuration by referencing a subscriber profile record accessed by the digital computer and referenced by a caller identification number.

[*12] 12. The method of claim 11 whereby the step of arranging includes using a formatted, screen interactive development program that uses a graphical interface providing visual manipulation of the vectors, objects, and events of each caller interface configuration.

[*13] 13. The voice processing system of claim 1, wherein:

said configuration means is further operable to configure a plurality of selectable interface configurations;

said storage means is further operable to store said selectable interface configurations; and

further comprising selection means for allowing each caller to select a particular interface configuration from among said selectable interface configurations.

[*14] 14. The method of claim 5, wherein:

said step of configuring comprises configuring a plurality of selectable interface configurations; and

said step of storing comprises storing the selectable interface configurations;

such that each caller can select a particular interface configuration from among the selectable interface configurations.

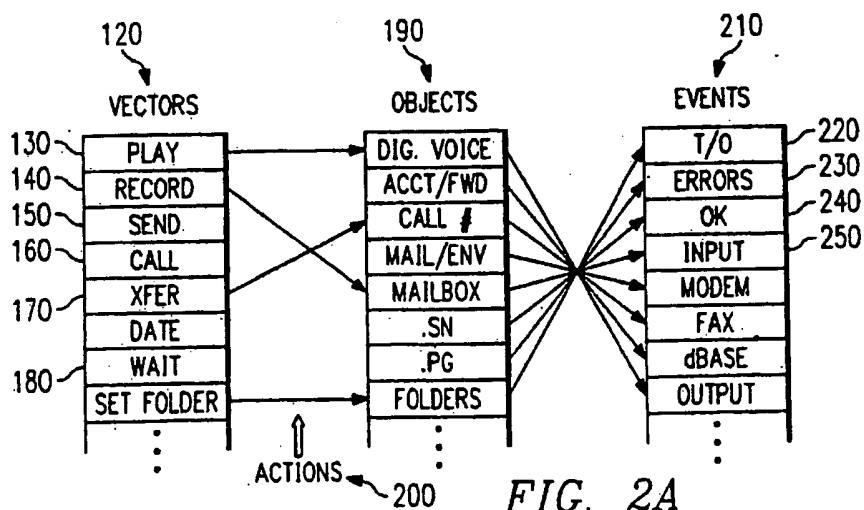
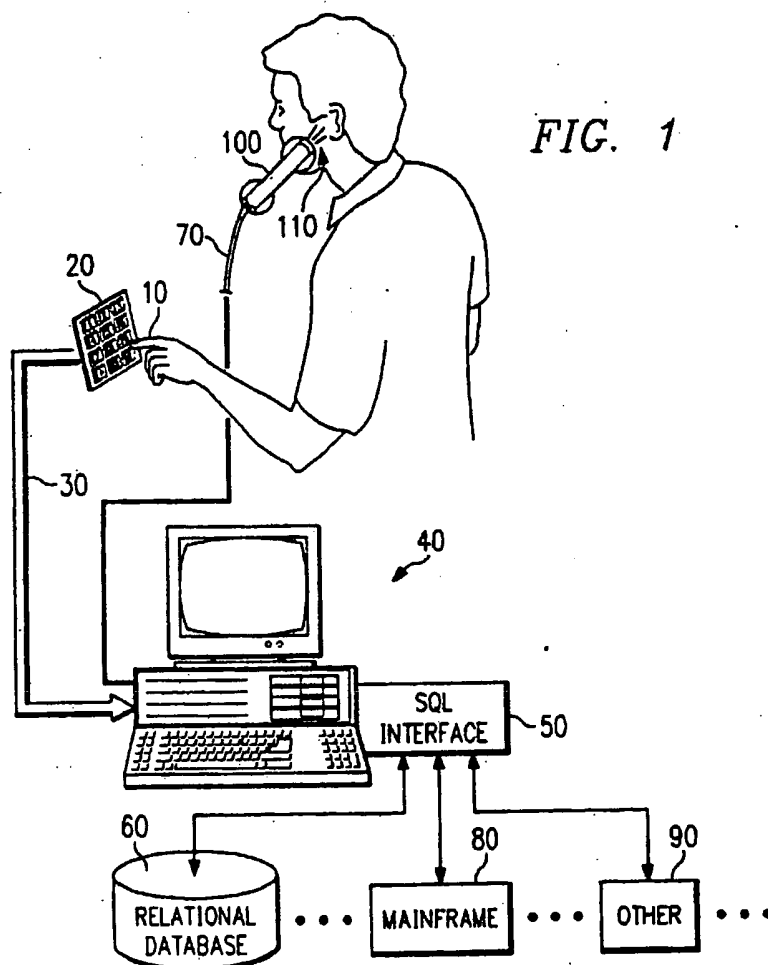
[*15] 15. The voice processing system of claim 1 wherein said configuration means includes a formatted, screen interactive development program that uses a graphical interface providing visual manipulation of the vectors, objects, and events of the caller interface configurations.

U.S. Patent

Sep. 7, 1993

Sheet 1 of 7

5,243,643



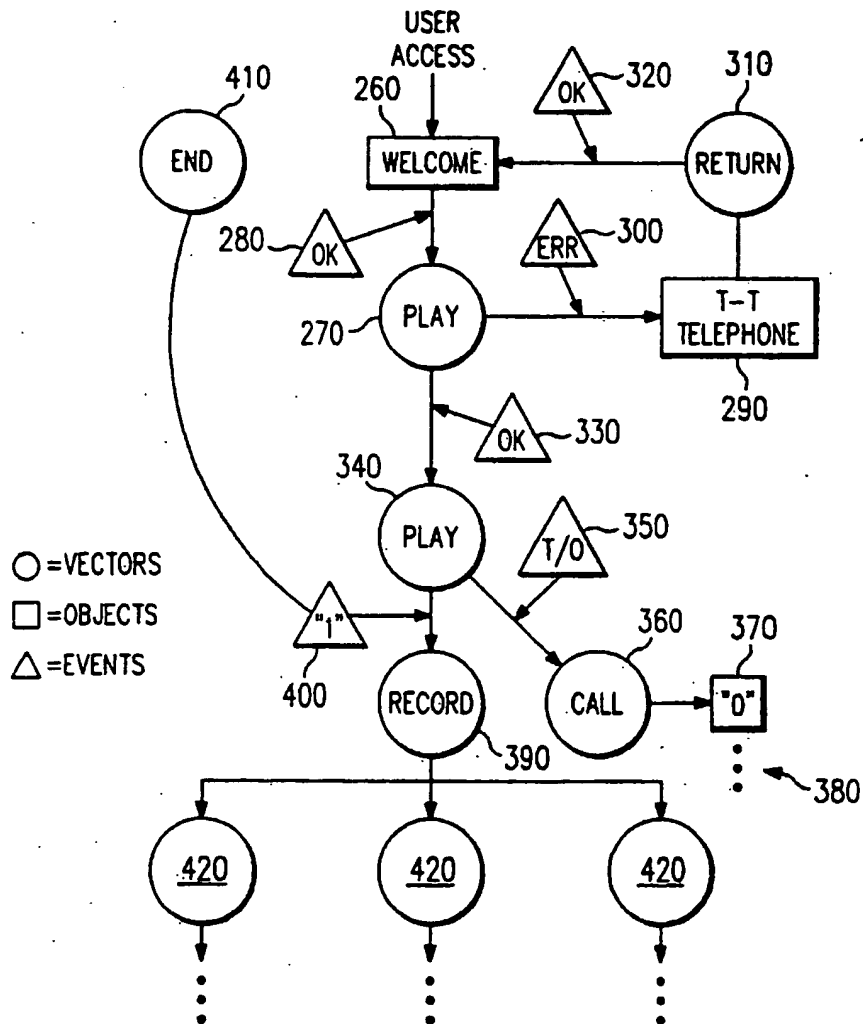


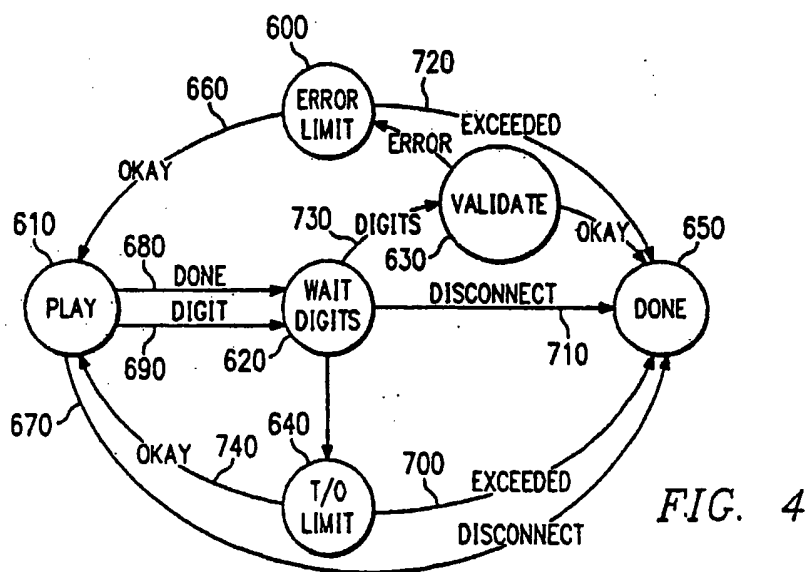
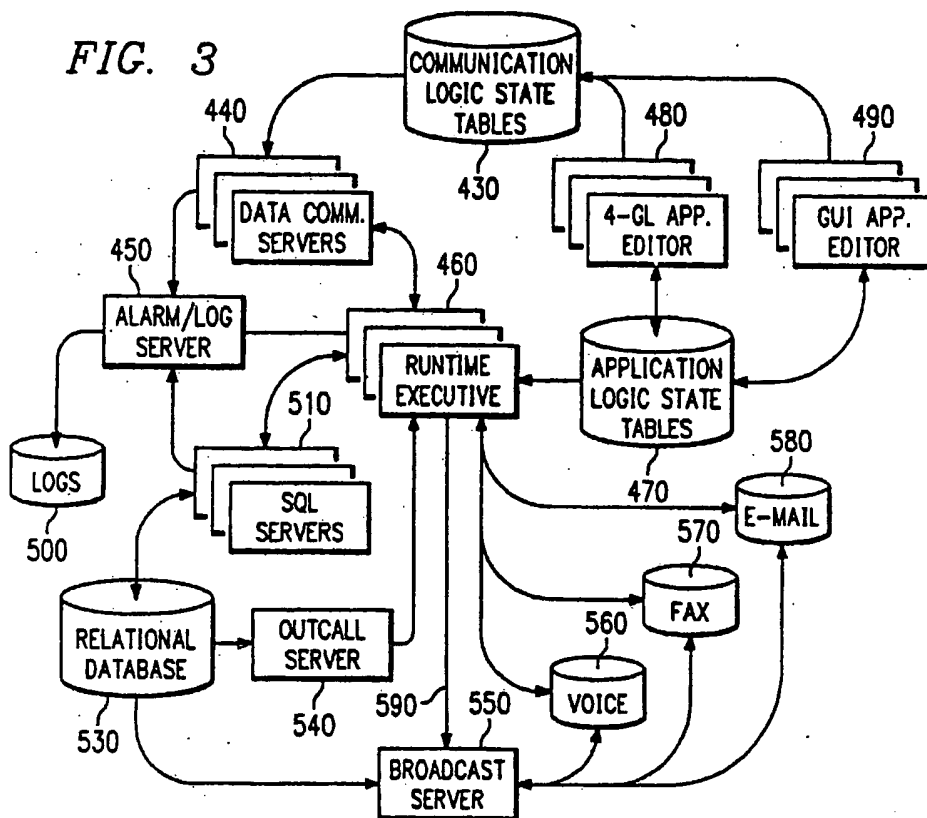
FIG. 2B

U.S. Patent

Sep. 7, 1993

Sheet 3 of 7

5,243,643



U.S. Patent

Sep. 7, 1993

Sheet 4 of 7

5,243,643

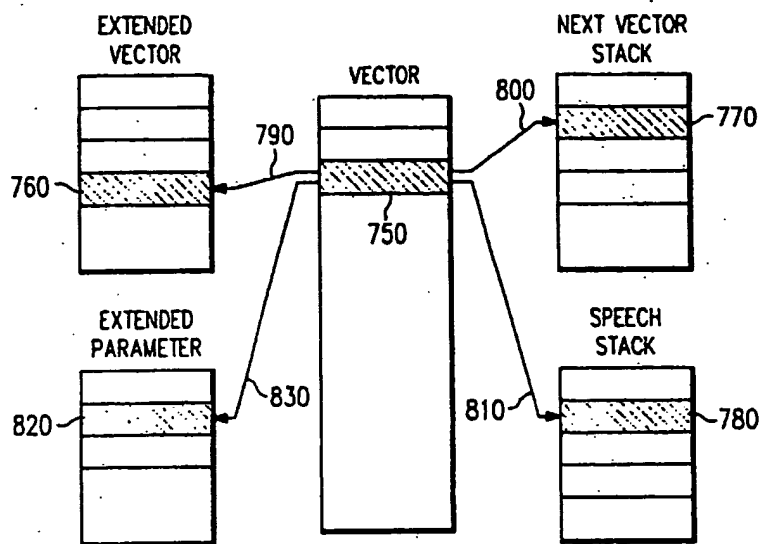


FIG. 5A

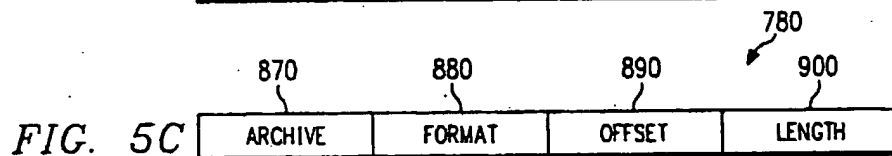
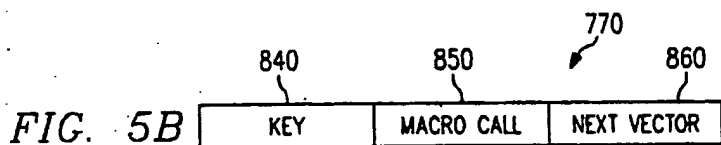


FIG. 5C

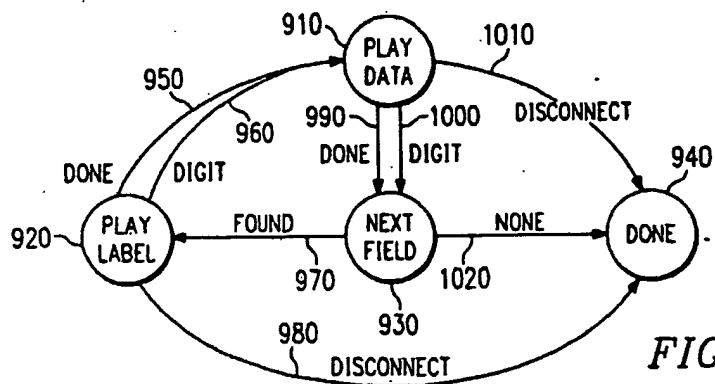
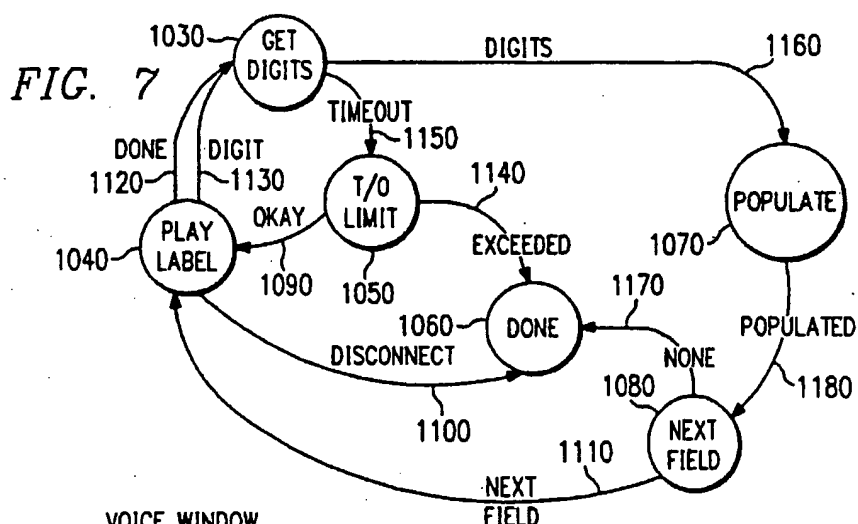


FIG. 6



VOICE WINDOW FIELD TYPES

1300	CHARACTER
1310	FILLER
1320	TIME STAMP
1330	UNIQUE FILE NAME
1340	FORMATTED DATE/TIME
1350	CONSTANT
1360	REFERENCE
1370	COPY
1380	SEQUENCE

FIG. 9A

VOICE WINDOW ENUNCIATION TYPES

1390	DATE
1400	MONEY
1410	NUMERIC
1420	PAIRED NUMERIC
1430	PHRASE
1440	TIME
1450	PERCENT
1460	NONE

FIG. 9B

VOICE WINDOW FIELD ATTRIBUTES

1480	VERIFY
1490	PARTIAL INPUT
1500	CONTINUE ON ERROR
1510	NON-VOLATILE

FIG. 9C

VOICE WINDOW SCHEMA

1190	WINDOW NAME
1200	FIELD NAME
1205	FIELD TYPE
1210	INPUT SIZE
1220	VOICE LABEL
1230	CROSS WINDOW NAME
1240	CROSS FIELD NAME
1250	VALIDATION FUNCTION NAME
1260	CONVERSION FORMAT
1270	ENUNCIATION TYPE
1280	INPUT DELIMITERS
1290	FIELD ATTRIBUTES

FIG. 8

COMPILED VECTOR ELEMENT

1520	VECTOR NAME
1530	TYPE
1540	SUB-TYPE
1550	PARAMETER
1560	EXTENDED PARAMETER
1570	EXTENDED VECTOR
1580	NEXT VECTOR STACK
1590	SPEECH STACK

FIG. 10

U.S. Patent

Sep. 7, 1993

Sheet 6 of 7

5,243,643

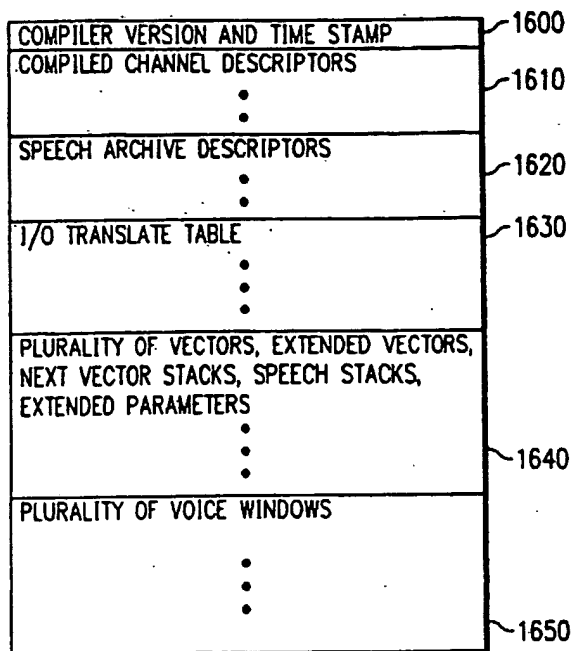


FIG. 11

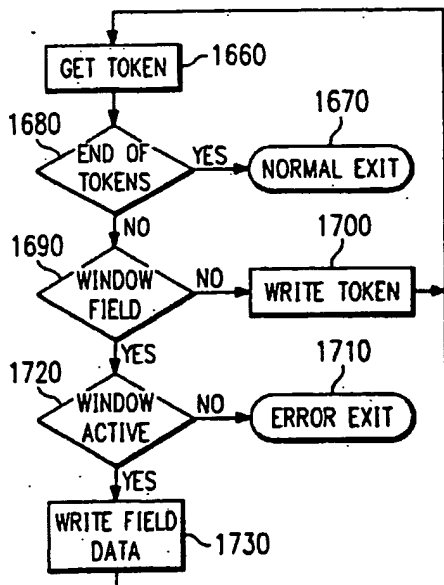


FIG. 12

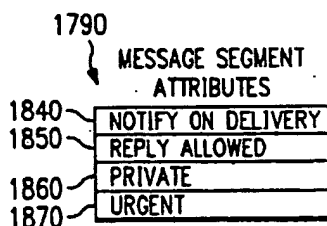


FIG. 13B

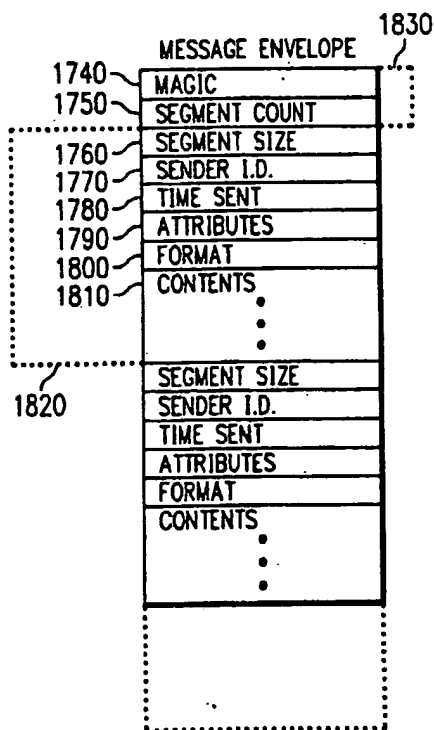


FIG. 13A

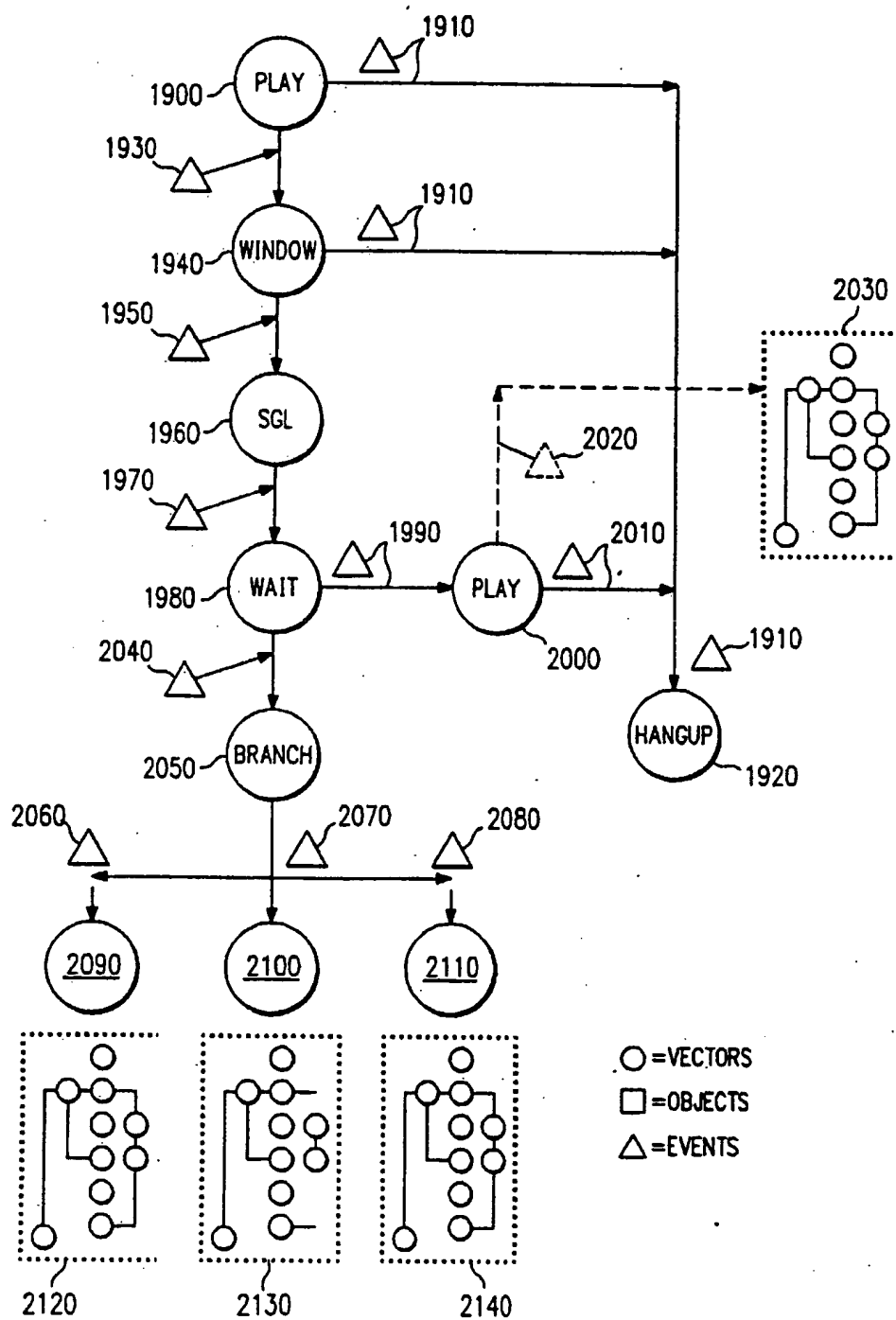


FIG. 14

Pat. No. 5737393 printed in FULL format.

5,737,393

<=2> GET 1st DRAWING SHEET OF 8

Apr. 7, 1998

Script-based interactive voice mail and voice response
system

INVENTOR: Wolf, Richard J., Crowley, Texas

ASSIGNEE-AT-ISSUE: AST Research, Inc., Irvine, California (02)

APPL-N0: 821,413

FILED: Mar. 21, 1997

REL-US-DATA:

Continuation of Ser. No. 508,391, Jul. 31, 1995 now abandoned

INT-CL: [6] H04M 1#64

US-CL: 379#88.13;

CL: 379;

SEARCH-FLD: 379#67, 88, 89, 201, 214, 97, 211, 212, 93, 95, 188, 96

REF-CITED:

U.S. PATENT DOCUMENTS		
4,594,476	6/1986	* Freeman 379#73
4,866,756	9/1989	* Crane 379#188
5,065,347	11/1991	* Pajak 379#159
5,243,643	9/1993	* Sattar 379#88
5,255,305	10/1993	* Satter 379#97
5,329,578	7/1994	* Brennan et al. 379#67
5,349,636	9/1994	* Irribarren 379#67
5,393,964	2/1995	* Hamilton 379#91
5,416,830	5/1995	* MacMillan, Jr. 379#97
5,416,831	5/1995	* Chewning 379#97
5,469,500	11/1995	* Satter et al. 379#201

OTHER PUBLICATIONS

"Repartee Version 7.1-Telanophy Feature Package Guide", Active Voice Corporation, pp. 1, 2, 53, 55-57, 59, 65 and 67-69, Mar. 1994.

PRIM-EXMR: Zele, Krista

ASST-EXMR: Wolinsky, Scott

LEGAL-REP: Knobbe, Martens, Olson & Bear, L.L.P.

CORE TERMS: menu, prompt, mailbox, user, caller, password, message, script,

display, queue, played, routine, dialog, box, greeting, enabling, menus, displayed, telephone, engine, mail, tab, telephone line, notification, icon, interactive, recorded, electrically, remote, hangup

ABST:

Method and a system for a user-customizable interactive voice mail/voice response system are disclosed. In a preferred embodiment, an interactive voice mail/voice response ("IVR") system of the present invention enables a user to build and operate custom IVR functions. The system comprises a voice menu system comprising a plurality of menus, each comprising a plurality of script records, and a scripting engine for providing event queuing functions to the voice mail system. When a menu is activated in response to an incoming call, the activated menu queues appropriate events with the script engine for playing a greeting to the telephone line and optionally requesting and verifying a password input by the caller. The menu then requests each record to queue events for providing a prompt indicating the key association and the purpose of the record. The engine processes the event queue and concurrently processes incoming keys, which are sent to the active script by the menu and then dispatched by the menu to the record with which the key is associated. The record responds with event requests appropriate to its implementation. This continues until the call is terminated. In one aspect of the invention, the script engine is connected to a script editing mechanism for enabling a user to create new menus, change the properties of existing menus, add records from an available inventory of record types and change the properties of existing records using a series of graphical user interface screens and dialog boxes designed for that purpose.

NO-OF-CLAIMS: 24

EXMPL-CLAIM: <=14> 12

NO-OF-FIGURES: 16

NO-DRWNG-PP: 8

PARCASE: This is a continuation of application Ser. No. 08/508,391 filed on Jul. 31, 1995, now abandoned.

SUM:

TECHNICAL FIELD

The invention relates generally to interactive voice response and voice mail systems and, more particularly, to a script-based method of authoring such systems.

BACKGROUND OF THE INVENTION

Various types of interactive voice response systems are well known in the art and typically comprise some means for prompting a caller who has dialed into the system to enter information and/or questions, often in response to a menu of voice prompts issued via the handset of the caller's telephone, wherein caller entries generally consist of DTMF key tones generated using the keypad of the caller's telephone. For example, the caller may be prompted to enter a "1" to inquire about an account balance, a "2" to speak to a customer service

representative, or a "0" to speak to the operator. Assuming the caller enters a "1" in response to the first series of prompts, the caller may be prompted to enter his or her account number, at which point, the system will determine the balance of the account associated with the entered account number and announce same to the caller.

Voice mail systems are typically designated as those in which a caller can leave a message for an intended recipient who is not available to take the call at the time the call is received in the system. The caller's message is stored in some type of central memory device, in which all messages received for all extensions on the system are stored, and is designated as being for a single extension. Only the designated recipient can retrieve the messages left for his or her extension. The set of messages designated as being for a single recipient is commonly referred to as the recipient's "voice mailbox," although the messages may be stored throughout the memory device.

Currently available interactive voice response and voice mail systems range high-end systems aimed at the large commercial consumers, which are typically complex systems custom designed to serve a variety of user-specified purposes, to lower-end systems aimed at the smaller commercial and individual consumers, which systems are typically less complex and capable of implementing fewer features. Moreover, although "lower end" systems have been developed which allow the user to customize certain voice mail and voice response features, such systems typically require a greater level of programming skill and knowledge than the average computer user is apt to possess.

Therefore, what is needed is an interactive voice mail/voice response system that is simple and straightforward enough for the average user to install, customize and operate.

SUMMARY OF THE INVENTION

The present invention, accordingly, provides a method and an interactive voice mail/voice response system that may be customized by a user to overcome or reduce disadvantages and limitations associated with prior methods and systems.

In a preferred embodiment, an interactive voice mail/voice response ("IVR") system of the present invention enables a user to build and operate custom IVR menus. The system comprises a voice menu system comprising a plurality of menus, each comprising a plurality of script record implementations, and a scripting engine for providing an event queuing mechanism to the menu records. The records in turn provide a sink mechanism for line events, such as DTMF key tones and telephone line state changes.

In operation, when a menu is activated in response to an incoming call, the menu queues any appropriate events with the script engine. These events typically play a greeting to the telephone line and optionally request and verify a password, which is input by the caller via the keypad of the telephone. The menu then requests each record to queue events. Typically these events provide a prompt indicating the key association and the purpose of the record. The engine then processes the event queue and concurrently processes incoming keys. Keys are sent to the active script by the menu and the menu, in turn, dispatches the key to the record with which the key is associated. The record then responds with event requests that are appropriate to its implementation.

claim 1

This continues until the caller hangs up or a record is invoked that terminates the call.

In one aspect of the invention, the script engine is connected to a script editing mechanism that allows the user to create new menus, change the properties of existing menus, add records from the available inventory of record types and change the properties of existing records quickly and easily via a series of graphical user interface screens and dialog boxes designed for that purpose.

A technical advantage achieved with the invention is that it provides a method of authoring IVR systems that are sophisticated and dynamic in nature without the use of complicated authoring systems, such as proprietary scripting languages, traditional programming languages or other complex techniques.

Another technical advantage achieved with the invention is that it provides a scripting engine that supports telephone line management and event queuing.

Yet another technical advantage achieved with the invention is that IVR responses are implemented in an object oriented manner, in which script records are subclasses of more basic IVR types.

Still another technical advantage achieved with the invention is the ability to present IVR systems as menus, or scripts, as a simple table of IVR objects.

Still another technical advantage achieved with the invention is that it provides the ability to reference menus from within another menu, thus achieving a versatile control mechanism presented to the user in a simple manner.

DRWDESC:
BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1A is a system block diagram of a personal computer ("PC") embodying features of the present invention.

FIG. 1B is a conceptual block diagram of the integrated voice mail/voice response ("IVR") system of the present invention.

FIG. 2 illustrates a main screen of the user interface of the IVR system of the present invention.

FIG. 3 illustrates a Menu Properties dialog box of the user interface of the IVR system of the present invention for creating and modifying menus.

FIG. 4 illustrates a Mailbox Properties dialog box of the user interface of the IVR system of the present invention for creating and modifying Mailbox records.

FIG. 5 illustrates a Menu Reference Properties dialog box of the user interface of the IVR system of the present invention for creating and modifying Menu Reference records.

FIG. 6 illustrates a Read Mailbox Properties dialog box of the user interface of the IVR system of the present invention for creating and modifying Read Mailbox records.

FIGS. 7, 8, 9A, 9B, and 10-14 are flowcharts illustrating the operation of the IVR system of the present invention.

DETDISC:

DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIG. 1A illustrates a personal computer ("PC") 10 embodying features of the present invention. In particular, as shown in FIG. 1, the PC 10 comprises a central processing unit ("CPU") 12, memory and storage devices, collectively designated by a reference numeral 14, a mouse 16, a keyboard 18, a display 19, and a microphone 20 interconnected via one or more computer buses, shown in FIG. 1 as a bus 22. At least one Telephony Application Programming Interface ("TAPI") compliant device 24 is provided on the bus 22 for connecting the components of the PC 10 to a telephone line (not shown) and for providing the PC 10 with standard telephone capabilities. It will be recognized by those skilled in the art that "TAPI," which was jointly developed by Microsoft Corporation of Redmond, Wash., and Intel Corporation of Santa Clara, Calif., defines an interface between Microsoft Windows applications and telephone devices for providing a standard way to build telephone capabilities into Windows software. Alternatively, the device 24 may be a voice modem. As will be described, instructions for implementing an integrated voice mail/voice response ("IVR") system 26 are stored in the memory/storage device 14 for execution by the CPU 12.

FIG. 1B is a conceptual block diagram of the IVR system 26 shown in FIG. 1A. In particular, the IVR system 26 comprises a voice menu system 100 that includes a plurality of menus 102. Each menu 102 includes a plurality of menu records 104. As will be described in greater detail below, the number of menus and records per menu is virtually unlimited and is established by the user. It should be noted, however, that the number of useful records is limited to the number of distinct DTMF tones that can be presented to the system, typically twelve, with a common telephone. The IVR system 26 further comprises an IVR script engine 105, which includes a telephone management system 106 for managing the device 24, an event queue management system 108 for queuing the appropriate events, as indicated by the menu records 104 of the current, or active, menu, with the script engine 105, and a script editing system 110 for enabling a user to add and delete menus 102 and menu records 104 from the voice menu system 100, as will be described.

In accordance with the features of the present invention, a user's use of the IVR system 26 is facilitated by the use of a plurality of graphical user interface ("GUI") screens and dialog boxes for display on the display 19. FIG. 2 illustrates a main screen 200 comprising a menu section 202a and a records section 202b. A plurality of menu icons 204, in this case, a "Main" icon and a "Remote" icon, are displayed in the menu section 202a. Upon selection of one of the menu icons 204 displayed in the menu portion 202a, a corresponding menu, or script, for the selected menu is displayed in the records section 202b. As shown in FIG. 2, the Main icon has been selected, resulting in the display in the records section 202b of the script 206. Each line of the displayed script 206 comprises an individual menu, or script, record and, for purposes to be described in detail below, each record of the script 206 includes a "Key" component, a "Parameter" component, a "Menu type" component, and a "Prompt" component.

As will be described in detail below, using the IVR system 26 of the present invention, the user is able to create, modify, and delete entire menus, such as Main or Remote, as well as one or more records within a menu. As will be recognized, a user can create as many different menus and corresponding scripts as he or she desires. It should be noted that, as used herein, "menu" and "script" may be used interchangeably to refer to a set of individual records.

To create a new menu, which will be represented by an icon similar to the icons 204 in the menu section 202a, the user accesses a "Menu Properties" dialog box, as shown in FIG. 3, through a series of appropriate menu picks, as will be recognized by those skilled in the art. From the Menu Properties dialog box, the user may select a Menu tab 300 to access a Menu page, which is shown in FIG. 3, to enter a name of the new menu, for example, "Basic Machine," a password to be associated with the menu which must be entered before a caller can access the menu, and an icon for representing the menu in the menu section 202a of the main screen 200 (FIG. 2). It should be noted that if a password is not entered on the Menu page, no password will be required and the menu will be freely accessible by callers. Similarly, although not shown, selection of a "Greeting" tab 302 results in the display of a Greeting page that enables the user to record a greeting for the menu. A typical greeting might be, for example, "Welcome to the XYZ company customer service facility."

As with all prompts described herein, the greeting is recorded by the user's typing the text of the greeting on the Greeting page (not shown) and then recording the audio greeting using the microphone 20. The recorded greeting will be stored as a sound file, most likely, a WAVE format file, designated as the greeting prompt for the corresponding menu, in this case, "Basic Machine." Alternatively, text-to-voice synthesis techniques may be used to convert the typed prompt to an audio prompt to be stored in the sound file for the prompt.

Selection of a "Password" tab 304 results in the display of a Password page (not shown), the format of which is similar to that of the Menu page shown in FIG. 3. Assuming that a password has been entered in the Menu page, the Password page may be used to record a password prompt to be played immediately before the greeting. A typical password prompt may be "Please enter your four character account number now." The password prompt is recorded in the same manner in which the greeting prompt is recorded, as described above. As previously indicated, if no password is entered on the Menu page (FIG. 3), it is not necessary to record a password prompt.

Selection of a "Bad Password" tab 306 results in the display of a Bad Password page (not shown), the format of which is similar to that of the Menu page shown in FIG. 3. As with the Password page, assuming a password has been entered on the Menu page (FIG. 3), the Bad Password page enables a user to record a bad password prompt to be played when a password entered by the caller in response to the password prompt is incorrect. A typical bad password prompt might be, for example, "The password you have entered is incorrect." At that point, the password prompt is replayed and the process is repeated until the caller enters the correct password or the call is terminated.

Finally, selection of a "Prompt" tab 304 results in the display of a Prompt page (not shown), the format of which is similar to that of the Menu page shown in FIG. 3. Using the Prompt page, the user may record a menu prompt for the current menu (i.e., Basic Machine) in the same manner in which the greeting prompt is recorded, as described above. A typical menu prompt might be "You

may choose from the following options at any time during this call." The menu prompt will be played once the caller has entered the correct password in response to the password prompt, if a password is required, or immediately after the greeting, if no password is required. After the menu prompt is played, the individual record prompts, which are created in a manner to be described in detail below, are played in the order shown in the script for the menu.

Referring again to FIG. 2, once a menu has been created, as described above with reference to FIG. 3, the user may create and add records to the menu by selecting the icon associated with the menu to be modified from the menu section 202a, which results in the script associated with the selected icon being displayed in the script section 202b. At that point, the user may modify the script by accessing a script record type menu (not shown) comprising a series of menu picks, each associated with an available type of record. For example, in the current embodiment, there are four types of script records, including a "Mailbox" record for enabling a caller to record a message, and then depositing the recorded message in the corresponding mailbox location, a "Menu Reference" record for enabling a caller to activate, or access, a second menu from a first menu, a "Read Mailbox" record for enabling a caller to read messages previously deposited in his or her mailbox, and a "Hangup" record for terminating the call; however, it should be recognized that any number of different types of records having various different properties may be implemented using the system 26.

Selection of the Mailbox menu pick from the script record type menu results in the display of a Mailbox Properties dialog box, as shown in FIG. 4, for use in creating a Mailbox-type record. Selection of a "Mailbox" tab 400 results in the display of a Mailbox page, as illustrated in FIG. 4, which enables the user to enter the name of the mailbox, in this case "Jean," a DTMF key to be associated with the mailbox, in this case, "1," for accessing the mailbox from the menu, and also an icon for representing the record if desired. Selection of a "Prompt" tab 402 results in the display of a Prompt page (not shown) for enabling the user to record a prompt to be played in connection with the record. In this case, a typical prompt might be "Press 1 to leave a message for Jean." Selection of "Pre-record prompt" tab 404 results in the display of a Pre-record prompt page (not shown) for enabling the user to record a prompt to be played after the record is selected by the remote caller by pressing the corresponding DTMF key, as designated on the Mailbox page (FIG. 4) and just before the incoming message is recorded. A typical pre-record prompt might be "At the sound of the tone, please leave a message for Jean."

Referring again to FIG. 2, it will be recognized that, for records created using the Mailbox Properties dialog box described above, the "Name" and "Key" entered on the Mailbox page (FIG. 4) will be respectively displayed as the "Parameter" and "Key" components of the associated record, the prompt entered on the Prompt page (not shown) will be displayed as the "Prompt" component of the record, and "Mailbox" will be displayed as the "Menu type" component of the record.

FIG. 5 illustrates a Menu Reference Properties dialog box for use in defining a Menu Reference-type record. A Menu Reference record enables a remote caller to reference, or access, another menu from within a first menu. For example, as shown in FIG. 2, the system may be set up to have a Main menu that is encountered by all callers when they call into the system, and a Remote menu that enables callers to access their own mailbox to retrieve their messages remotely. In this situation, the Remote menu could be referenced by, and

thereby accessible through, a record within the Main menu. For example, in the script 206 shown in FIG. 2, pressing "9" enables the caller to access the Remote menu. Similarly, the Main menu might be referenced by a record in the Remote menu such that callers could return to the Main menu from the Remote menu after listening to their messages, if they so desired.

Accordingly, selection of a Menu tab 500 of the Menu Reference Properties dialog box results in the display of a Menu page, as shown in FIG. 5, for enabling the user to select from among previously created menus and assign a key for accessing the selected menu from the current menu. Similarly, selection of a "Prompt" tab 502 results in the display of a Prompt page (not shown) similar to the Menu page (FIG. 5) for enabling the user to record a prompt to be associated with the record, for example, "Press 9 to access options remotely." Referring again to FIG. 2, it will be recognized that for records created using the Menu Reference Properties dialog box shown in FIG. 5, the "Key" and menu selected on the Menu page are respectively displayed as the "Key" and "Parameter" components of the corresponding record, the prompt entered on the Prompt page is displayed as the "Prompt" component of the record, and "Menu reference" is displayed as the "Menu type" component of the record.

FIG. 6 illustrates a Read Mailbox Properties dialog box for use in creating a record for enabling a user to retrieve messages from his or her mailbox. It should be apparent that this type of record would typically be included in a "Remote" menu. Referring to FIG. 6, selection of a "Mailbox" tab 600 results in the display of a Mailbox page, as illustrated in FIG. 6, for enabling a user to select from among mailboxes that have previously been created using the Mailbox dialog box (FIG. 3), assign a key to the record and optionally assign a password for accessing the associated mailbox. Selection of a "Prompt" tab 601 results in the display of a Prompt page (not shown) for enabling the user to record a prompt for the record. A typical prompt might be, for example, "Press 1 to access Billy's mailbox." Similarly, selection of an "Introduction" tab 602 results in the display of an Introduction page (not shown) for enabling the user to select a format for an introduction to be played to the remote caller before any messages are played. A typical introduction might be, for example, "You have # messages," where # represents the number of messages as determined by the system 26. Selection of a "Message Header" tab 604 results in the display of a Message Header page (not shown) for enabling the user to select a format for a message header to be played before each individual message, for example, "Message 2 of 4 from John Smith on July 5 at 12:00 P.M." Selection of a "Completion" tab 606 results in the display of a Completion page for enabling the user to record or select a completion message for indicating that all of the messages have been played.

Referring again to FIG. 2, for each record created using the Read Mailbox Properties dialog box shown in FIG. 6, the "Mailbox" and "Key" entries are respectively displayed as the "Parameter" and "Key" components of the record, the prompt recorded using the Prompt page is displayed as the "Prompt" component of the record, and "Read Mailbox" is displayed as the "Menu type" component of the record.

Although not shown, it should be recognized that a Hangup Properties dialog box for use in creating a record for enabling a user to terminate the call by pressing a key would also be provided. In particular, a Hangup Properties dialog box would enable a user to record a prompt for and assign a key to the record, in a manner similar to that described above with reference to FIGS. 4, 5 and

6.

Referring again to FIG. 2, for each record created using the Hangup Properties dialog box (not shown), it will be recognized that the "Key" component will display the key associated with the record, the "Parameter" and "Menu type" components will each display "Hangup" and the "Prompt" component will display the prompt recorded using the Hangup Properties dialog box.

As previously indicated, the user can create any number of different menus comprising any number of records for controlling the operation of the IVR system 26. Referring to FIG. 2, assuming the Main menu is set as the default menu, i.e., the menu that is initially accessed when a caller calls into the system 26, the operation of the system 26 in response to an incoming call will now be described with reference to FIGS. 7-14. Referring briefly to FIG. 1B, as previously indicated, the scripting engine 105 provides telephone line management and an event queuing mechanism to the menu records 104. The records 104, in return, provide a sink mechanism for line events, such as receipt of DTMF key tone tones resulting from an caller's depression of keys and line state changes, such as a hang up. The engine can be notified of a script, or menu, change by a menu record 104 action, thereby giving the effect of nested menus and other complicated presentations without the user of conventional programming techniques on the part of the script, or menu, author.

When one of the menus 102 is activated, the menu queues any appropriate events with the scripting engine 105. In particular, the queuing of events involves creating an object for performing the specified function. These events typically play a greeting to the telephone line and optionally request and verify a password input using the DTMF keys. It will be recognized that whether or not a greeting is played, a password is requested, etc., will depend on how the active menu has been set up by the user, as shown and described with reference to FIG. 3. The menu then requests each record to queue events. Typically these events provide a prompt indicating the key association and the purpose of the record. Again, the events queued by each record will be determined by how that record was set up by the user, as shown and described with reference to FIGS. 4-6. The engine 105 processes the event queue and concurrently processes DTMF key tones. Keys are sent to the active menu by the engine 105, which in turn dispatches the key to the record with which the key is associated. That record then responds with events that are appropriate to its implementation. This continues until the caller hangs up or a record is invoked that terminates the call.

Referring now to FIG. 7, execution begins in step 700 upon receipt by the system 26 of a notification that an action performed external to the system 26 is complete. Upon receipt of a notification in step 700, execution proceeds to step 702, in which a determination is made whether the notification corresponds to an audio event completion notification. In particular, an audio event completion notification will be generated by the operating system when an audio event, such as the playing of a prompt to a caller or the recording or retrieval of a message by a caller, has been completed. If an audio event completion notification has been received, execution proceeds to step 704, in which the audio device that performed the audio event is closed, and then to step 706, in which the event is deleted from the queue. Execution then proceeds to step 708, in which a "play next event" routine, as shown and described with reference to FIG. 8, is executed.

Referring to FIG. 8, execution of the play next event routine begins in step 800. In step 802, a determination is made whether the first event in the queue is busy; that is, whether the event is being executed. It should be noted that, because the queue is implemented as a "first-in-first-out" queue, the "first event" in the queue refers to the top event in the queue, i.e., the oldest event therein remaining to be processed. If in step 802 it is determined that the first event is not busy, execution proceeds to step 804. In step 804, the first event is "requested." For example, in step 804, if the event is prompt, the requesting of the event causes the prompt to be played to the user; if the event is a timeout, the requesting of the event causes the operating system to schedule a timer; if the event is a record message event, the requesting of the event causes an incoming message to be recorded. Because the requesting of events will be understood to those skilled in the art of object oriented programming, it will not be described in further detail herein. After the event is requested in step 804, execution returns to the step that invoked the routine in step 806. If in step 802 it is determined that the first event is busy, execution proceeds directly to step 806.

Referring again to FIG. 7, if in step 702, it is determined that the notification is not an audio event completion notification, execution proceeds to step 710, in which a determination is made whether it is a notification that the phone is ringing. If so, execution proceeds to step 712, in which the system 26 answers the call, and then to step 714, in which a "play script" routine is executed, as described with reference to FIG. 9A.

Referring now to FIG. 9A, execution of the play script routine begins in step 900. In step 901, the current script, or menu, is designated as the active script. For example, as described above with reference to FIG. 2, the Main menu will most likely be the initial active menu. In step 902, a determination is made whether a password is required for the active menu. If so, execution proceeds to step 904, in which a "queue events" routine, as will be described in detail with reference to FIG. 9B, is executed to queue a gather digits event for processing by the engine 105. In step 906, the queue events routine (FIG. 9B) is executed to queue a password prompt. If in step 902, it is determined that no password is required, execution proceeds to step 908, in which the queue events routine (FIG. 9B) is executed to queue the greeting for the active menu, and then to step 910, in which the queue events routine (FIG. 9B) is executed to queue the menu prompt for the active menu. Execution then proceeds to step 912, in which a determination is made whether there is another record in the menu to be processed. If so execution proceeds to step 914, in which the queue event routine (FIG. 9B) is executed to queue the prompt for the next record, and then returns to step 912. If in step 912 it is determined that no records remain to be processed, execution proceeds to step 916, in which the queue event routine (FIG. 9B) is executed to queue a timeout event and then to step 918, in which the process is terminated. Similarly, upon completion of step 906, execution proceeds directly to step 918.

Referring now to FIG. 9B, the queue events routine will be described in greater detail. Execution of the routine begins in step 930. In step 932, an event object of the appropriate type is created, it being understood that the type of event object queued in step 932 will depend on the type of event, e.g., prompt, timeout, etc., to be queued. In step 934, the event object is added to the end of the list, or queue. In step 936, the play next event routine (FIG. 8) is executed. In step 938, execution returns to the calling step.

Referring again to FIG. 7, if in step 710, it is determined that the phone is not ringing, execution proceeds to step 716, in which a determination is made whether a DTMF key tone has been received. If so, execution proceeds to step 718, in which any currently busy event is stopped and all previously queued events are purged from the queue and then to step 720, in which a "play record of key" routine is executed, as described in detail with reference to FIG. 10.

Referring to FIG. 10, execution of the play record of key routine begins in step 1000. In step 1002, a determination is made whether the first record in the script is the record identified by the received key tone, it being understood that this step is performed by comparing the entered key to the key specified in the record in the "Key" component. If not, execution proceeds to step 1004, in which a determination is made whether there are more records in the active menu. If so, execution returns to step 1002 and the next record is compared. If in step 1004 there are no more records, a default record, typically a record for which a key has not been designated, is selected in step 1008 and played in step 1010, as described with reference to FIG. 12, 13 or 14, depending on whether the record is a Mailbox, Read Mailbox, or Menu Reference type record. If the record is a Hangup type record, a hangup event notification is generated (see step 728, FIG. 7). Similarly, if in step 1002, a match is found, execution proceeds directly to step 1010 and the matching record is played. The routine terminates execution in step 1012.

Referring again to FIG. 7, if in step 716 a DTMF key tone was not received, execution proceeds to step 722. In step 722, a determination is made whether the notification was a timeout generated responsive to the timing out of a timer set in step 1016 (FIG. 10). If so, execution proceeds to step 724, in which any currently busy event is stopped and the events are purged from the queue, and then to step 726, in which the play record of key routine, described above with reference to FIG. 10, is executed. The key associated with the timeout event is one that is not associated with any record in the active script.

If in step 722, it is determined that a timeout has not been received, execution proceeds to step 728, in which a determination is made whether a hangup occurred. If so, execution proceeds to step 730, in which any currently busy events are stopped and the events are purged from the queue; otherwise, execution proceeds to step 732. In step 732, a determination is made whether the event was a gather digits notification. If so, execution proceeds to step 734 in which a "consume digits" routine, described in detail with reference to FIG. 11, is executed.

Referring to FIG. 11, execution of the consume digits routine begins in step 1100. In step 1102, a determination is made whether the digits entered match the appropriate password. If so, execution returns to step 901 (FIG. 9A); otherwise, execution returns to step 908 (FIG. 9A).

Referring again to step 1010 (FIG. 10), assuming the record to be played is a Mailbox record, a "play mailbox record" routine, as shown and described with reference to FIG. 12, is executed to play the record. Referring to FIG. 12, execution of the play mailbox record routine begins in step 1200. In step 1202, the prerecord prompt, if any, for the record is queued. In step 1204, a unique filename is constructed for storing the message. In step 1206, a record file event for recording the message in the file is queued. Execution then terminates in step 1208.

If in step 1010 (FIG. 10), the record to be played is a Read Mailbox record, a "play read mailbox record" routine, as shown and described with reference to FIG. 13, is executed to play the record. Referring to FIG. 13, execution of the play read mailbox record routine begins in step 1300. In step 1302, a determination is made whether there are any messages in the mailbox designated by the record. If so, execution proceeds to step 1304, in which the introduction prompt for the record is queued. In step 1306, a determination is made whether there is another message. If so, execution proceeds to step 1308, in which the message header for the record is queued, and then to step 1310, in which the message filename is obtained. In step 1312, a play file audio event is queued, and then execution returns to step 1306. If in step 1306, it is determined that there are no more messages, execution proceeds to step 1314, in which the completion prompt for the record is queued. Execution then terminates in step 1316. Similarly, if in step 1302 it is determined that there are no more messages, execution proceeds directly to step 1316.

If in step 1010 (FIG. 10), the record to be played is a Menu Reference record, a "play menu reference record" routine, as shown and described with reference to FIG. 14, is executed to play the record. Referring to FIG. 14, execution of the play menu reference record routine begins in step 1400. In step 1402, the referenced menu is activated and execution terminates in step 1404.

As described above, the present invention provides a method of authoring IVR systems that are sophisticated and dynamic in nature without the use of complicated authoring systems, such as proprietary scripting languages, traditional programming languages, or other complex techniques. Because the invention displays the each IVR system menu as a simple table with "point and click" editing, the task of creating and modifying a menu is greatly simplified. Accordingly, complex menu sequences can be implemented by requesting script records that perform control features in a straightforward manner.

It is understood that the present invention can take many forms and embodiments, the embodiments shown herein are intended to illustrate rather than limit, the invention, it being understood that variations may be made without departing from the spirit of the scope of the invention. For example, records could be easily implemented to play a particular message, i.e., a "memo," to look up values from a database based on DTMF codes and/or CALLERID information, faxback documents, forward messages, provide pager notification, and so on. Moreover, prompts could be recorded using any number of different types of technology, including automatic text-to-voice simulation for converting the text prompt to an audio prompt, or using prerecorded prompts alone or in combination with user-recorded prompts.

Although illustrative embodiments of the invention have been shown and described, a wide range of modification, change and substitution is intended in the foregoing disclosure and in some instances some features of the present invention may be employed without a corresponding use of the other features. Accordingly, it is appropriate that the appended claims be construed broadly and in a manner consistent with the scope of the invention.

CLAIMS: What is claimed is:

[*1] 1. A method of implementing a user-definable interactive voice mail/voice response (IVR) system on a personal computer (PC) having a display, the method comprising user-implemented steps of:

creating at least one menu for implementing a plurality of IVR functions selected by a user, said creating including defining properties of said at least one menu using a menu dialog box displayed on a display of said PC;

adding at least one menu record for implementing an IVR function to said at least one menu, said adding comprising:

selecting a type of menu record to be added; and

defining properties of said menu record using a menu record dialog box displayed on said display in response to selecting said menu record type; and

graphically representing on said display said at least one menu including said at least one menu record.

[*2] 2. The method of claim 1 wherein said properties of said at least one menu include one or more of a name of said at least one menu, a password associated with said at least one menu, an icon for representing said at least one menu on a main screen display, and at least one prompt to be played to a caller over a telephone line to which said PC is electrically connected.

[*3] 3. The method of claim 2 further comprising recording said at least one prompt and representing a script of said at least one prompt on said display.

[*4] 4. The method of claim 1 wherein said selected menu record type is a mailbox record and wherein said properties of said menu record include one or more of a mailbox name, a DTMF key associated with said mailbox record for accessing said mailbox record and at least one prompt to be played to a caller via a telephone line to which said PC is electrically connected.

[*5] 5. The method of claim 4 further comprising recording said at least one prompt.

[*6] 6. The method of claim 1 wherein said selected menu record type is a menu reference record and wherein said properties of said menu record include one or more of a referenced menu name, a DTMF key associated with said menu reference record for accessing said menu reference record and a prompt to be played to a caller via a telephone line to which said PC is electrically connected.

[*7] 7. The method of claim 6 further comprising recording said prompt.

[*8] 8. The method of claim 1 wherein said selected menu record type is a read mailbox record and wherein said properties of said menu record include one or more of a mailbox name, a password associated with said read mailbox record, a DTMF key associated with said read mailbox record, and at least one prompt to be played to a caller via a telephone line to which said PC is electrically connected.

[*9] 9. The method of claim 8 further comprising recording said at least one prompt.

[*10] 10. The method of claim 1 wherein said selected menu record type is a hangup record and wherein said properties of said menu record include a DTMF

key associated with said hangup record and a prompt to be played to a caller via a telephone line to which said PC is electrically connected.

[*11] 11. The method of claim 10 further comprising recording said at least one prompt.

[*12] 12. A method of operating an interactive voice mail and voice response (IVR) system on a personal computer (PC) having a display and electrically connected to a telephone line, the IVR system comprising at least one menu for implementing a plurality of user-selected IVR functions in an order defined by a user and comprising a plurality of menu records each having associated therewith one of said user-selected IVR functions, the method comprising the steps of:

(a) presenting on said display a graphical representation of said at least one menu;

(b) said user selecting a menu graphically represented on said display by selecting said graphical representation thereof;

(c) responsive to detection of a ringing signal on said telephone line, answering said telephone line and activating said selected menu;

(d) playing at least one prompt associated with said selected menu to a caller over said telephone line;

(e) for each menu record in said selected menu, playing a first prompt associated with said menu record for instructing said caller regarding how to initiate an IVR function associated with said menu record; and

(f) responsive to receipt of a DTMF tone associated with one of said menu records over said telephone line, interrupting execution of step (e) and performing an IVR function associated with said one menu record.

[*13] 13. The method of claim 12 wherein step (d) further comprises:

prompting said caller to enter a password;

determining whether said entered password is valid for said selected menu; and

responsive to a determination that said entered password is not valid, repeating said prompting and determining until a valid password is entered or until a hangup is detected on said telephone line.

[*14] 14. The method of claim 12 wherein said one menu record comprises a mailbox type record and wherein said performing an IVR function associated with said one menu record comprises:

prompting said caller to record a message;

recording said message; and

storing said recorded message in a mailbox associated with said mailbox-type record.

[*15] 15. The method of claim 12 wherein said one menu record comprises a menu reference type record and wherein said performing an IVR function associated with said one menu record comprises:

activating a second menu identified by said menu reference type record; and
repeating steps (d) through (f) for said second menu.

[*16] 16. The method of claim 12 wherein said one menu record comprises a read mailbox type record and wherein said performing an IVR function associated with said one menu record comprises:

prompting said caller to enter a password;
determining whether said password is valid;

responsive to a determination that said password is not valid, repeating said prompting and determining until a valid password is entered; and

responsive to a determination that said password is valid, playing messages from a mailbox associated with said read mailbox type record to said caller via said telephone line.

[*17] 17. The method of claim 12 wherein said one menu record comprises a hangup type record and wherein said performing an IVR function associated with said one menu record comprises:

terminating communications with said caller.

[*18] 18. A user-definable interactive voice mail/voice response (IVR) system for implementation on a personal computer (PC) having a display, the system comprising:

means for creating at least one menu for implementing a user-specified set of IVR functions selected by a user, said means for creating comprising a menu dialog box displayed on a display of said PC for enabling said user to define properties of said at least one menu;

means for adding at least one menu record for implementing an IVR function to said at least one menu, said means for adding further comprising:

means for enabling said user to select a type of menu record to be added to said at least one menu; and

a menu record dialog box associated with said selected menu record type and displayed on said PC display for enabling said user to define properties of said at least one menu record; and

means for graphically representing on said display in response to selecting said menu record type said at least one menu including said at least one menu record.

[*19] 19. The system of claim 18 wherein said properties of said at least one menu include one or more of a name of said at least one menu, a password associated with said at least one menu, an icon for representing said at least

one menu on a main screen display, and at least one prompt to be played to a caller over a telephone line to which said PC is electrically connected.

[*20] 20. The system of claim 19 further comprising means for recording said at least one prompt and means for representing a script associated with said at least one prompt on said display.

[*21] 21. The system of claim 18 wherein said selected menu record type is a mailbox record and wherein said properties of said menu record include one or more of a mailbox name, a DTMF key associated with said mailbox record for accessing said mailbox record and at least one prompt to be played to a caller via a telephone line to which said PC is electrically connected.

[*22] 22. The system of claim 18 wherein said selected menu record type is a menu reference record and wherein said properties of said menu record include one or more of a referenced menu name, a DTMF key associated with said menu reference record for accessing said menu reference record and a prompt to be played to a caller via a telephone line to which said PC is electrically connected.

[*23] 23. The system of claim 18 wherein said selected menu record type is a read mailbox record and wherein said properties of said menu record include one or more of a mailbox name, a password associated with said read mailbox record, a DTMF key associated with said read mailbox record, and at least one prompt to be played to a caller via a telephone line to which said PC is electrically connected.

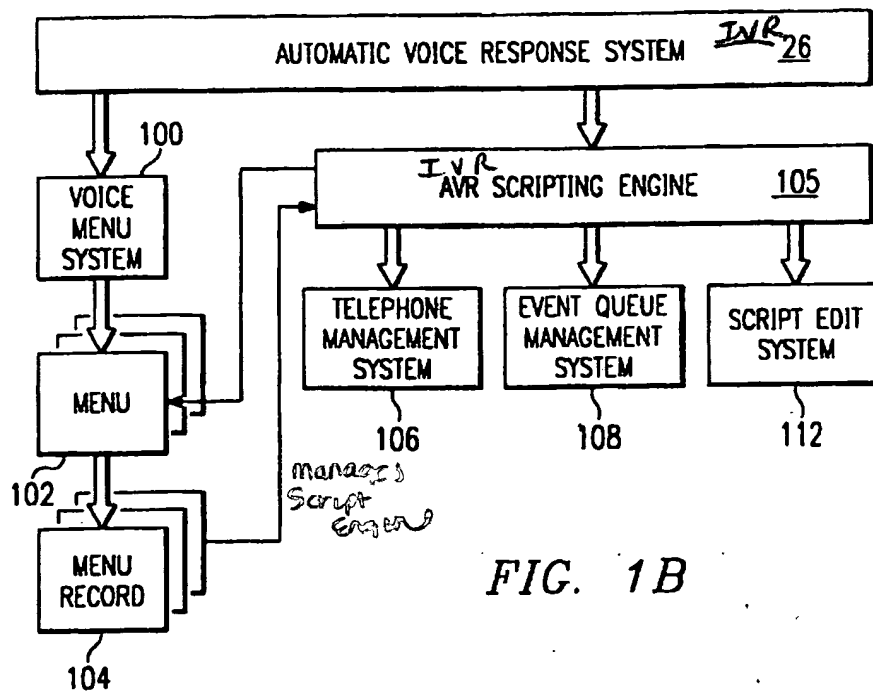
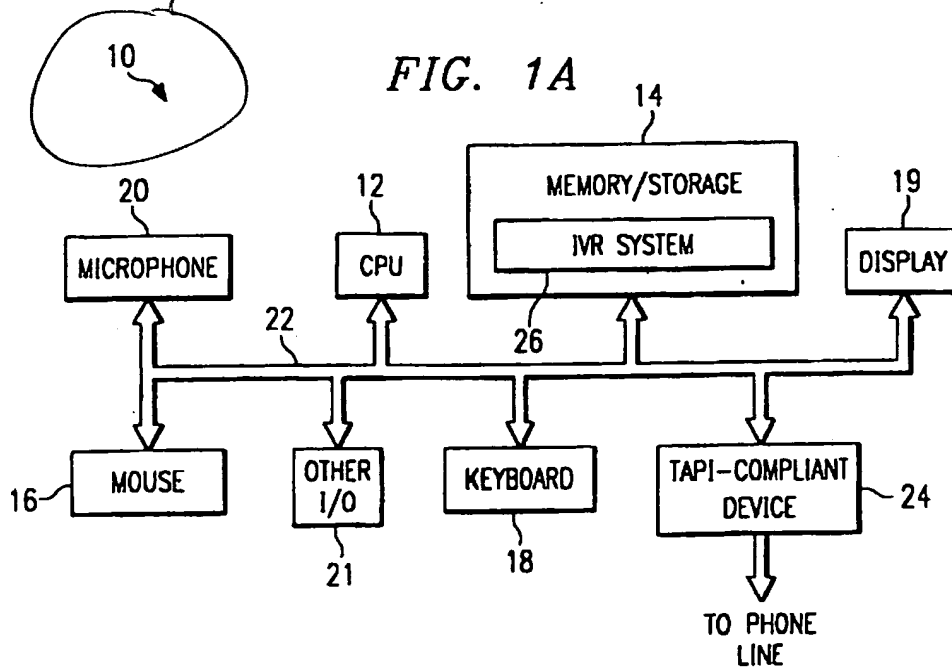
[*24] 24. The system of claim 18 wherein said selected menu record type is a hangup record and wherein said properties of said menu record include a DTMF key associated with said hangup record and a prompt to be played to a caller via a telephone line to which said PC is electrically connected.

U.S. Patent

Apr. 7, 1998

Sheet 1 of 8

5,737,393



U.S. Patent

Apr. 7, 1998

Sheet 2 of 8

5,737,393

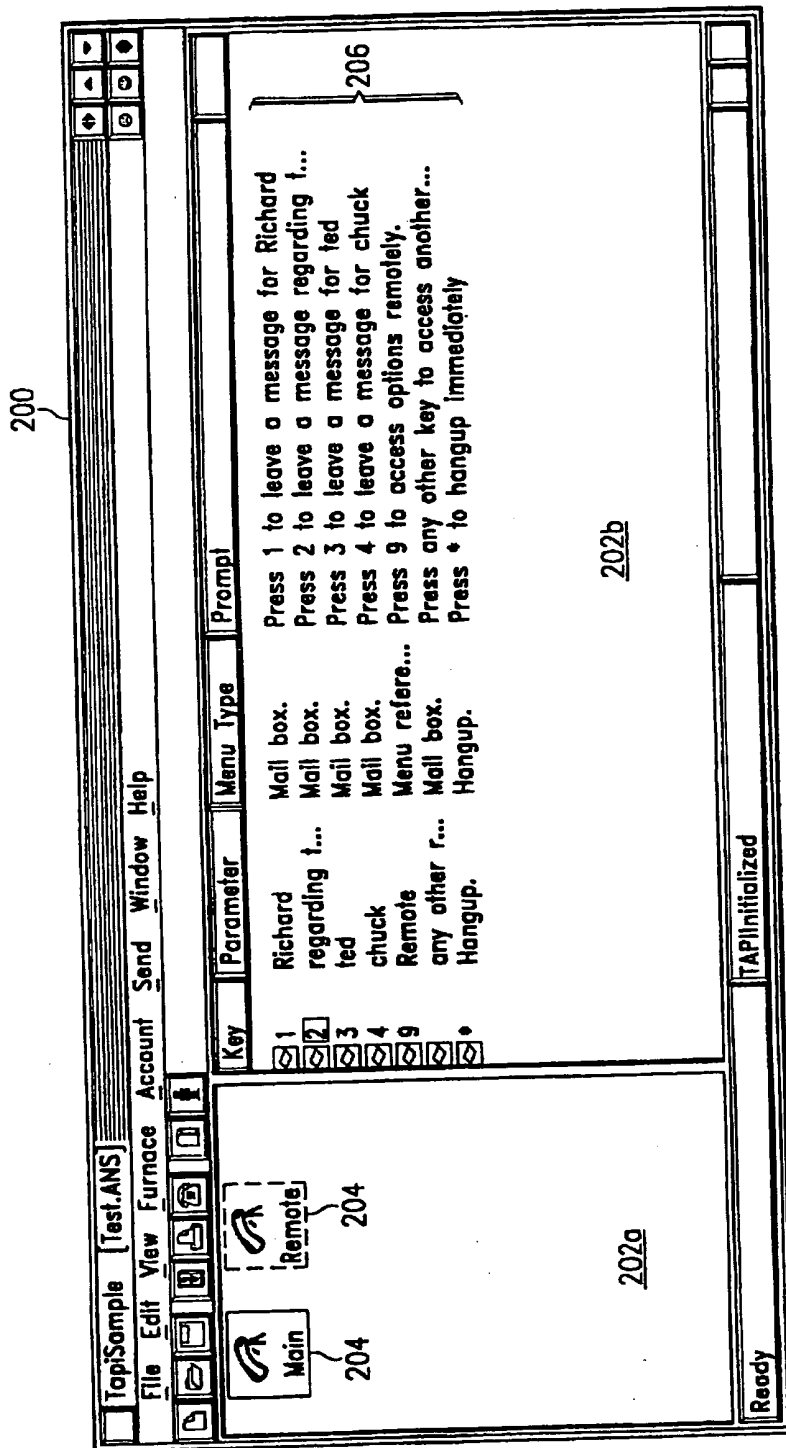


FIG. 2

U.S. Patent

Apr. 7, 1998

Sheet 3 of 8

5,737,393

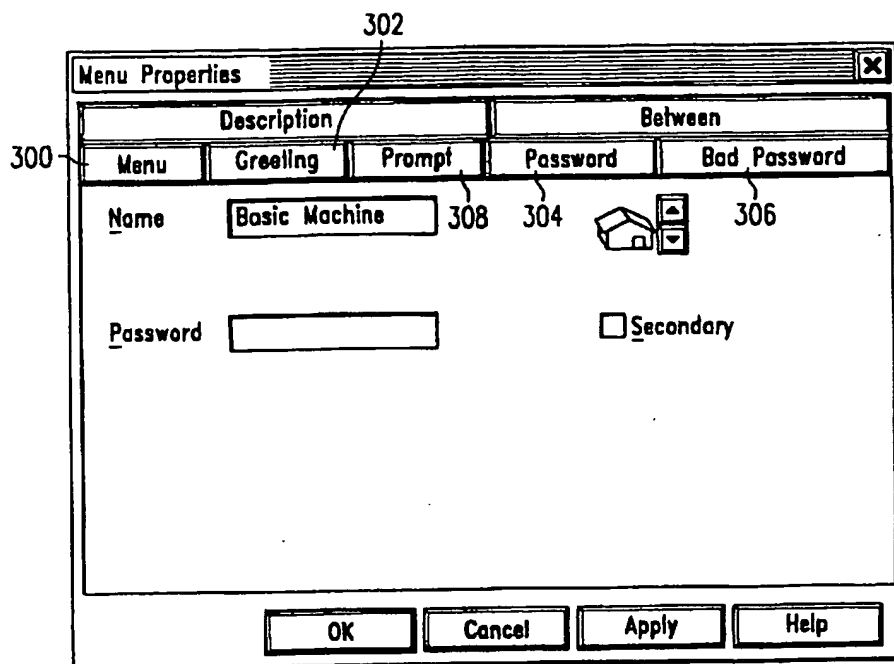


FIG. 3

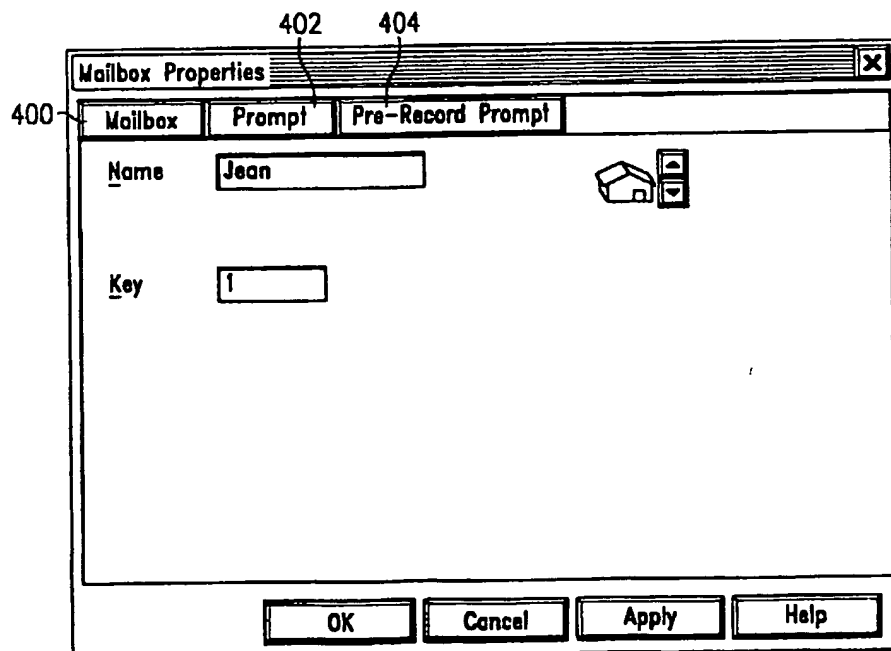


FIG. 4

U.S. Patent

Apr. 7, 1998

Sheet 4 of 8

5,737,393

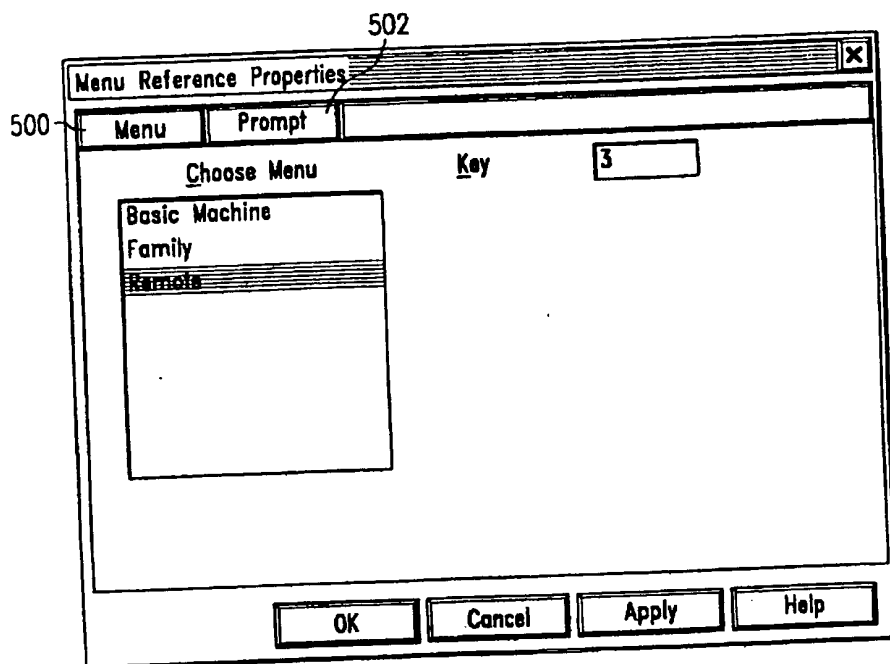


FIG. 5

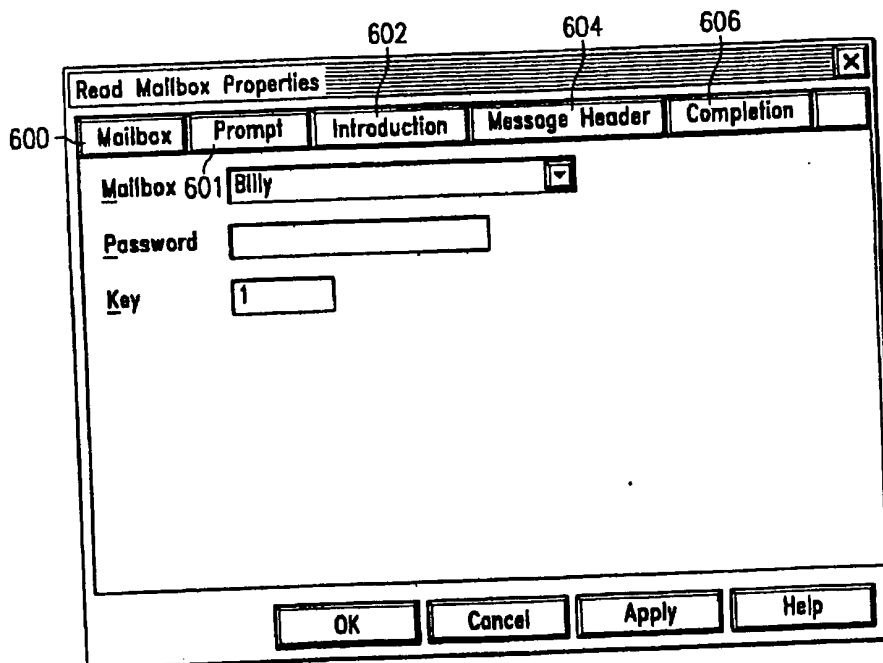


FIG. 6

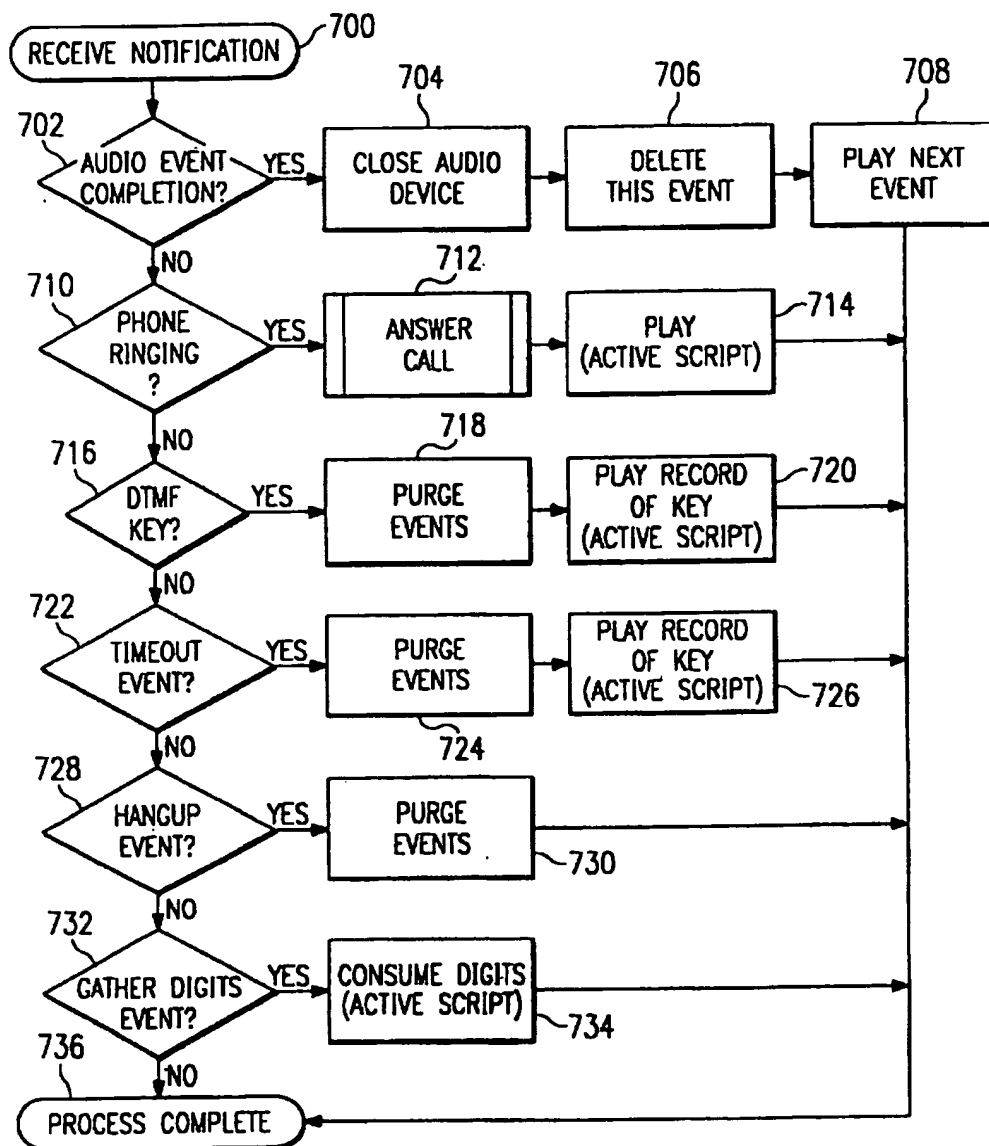


FIG. 7

U.S. Patent

Apr. 7, 1998

Sheet 6 of 8

5,737,393

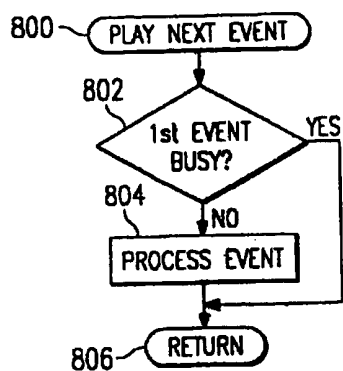


FIG. 8

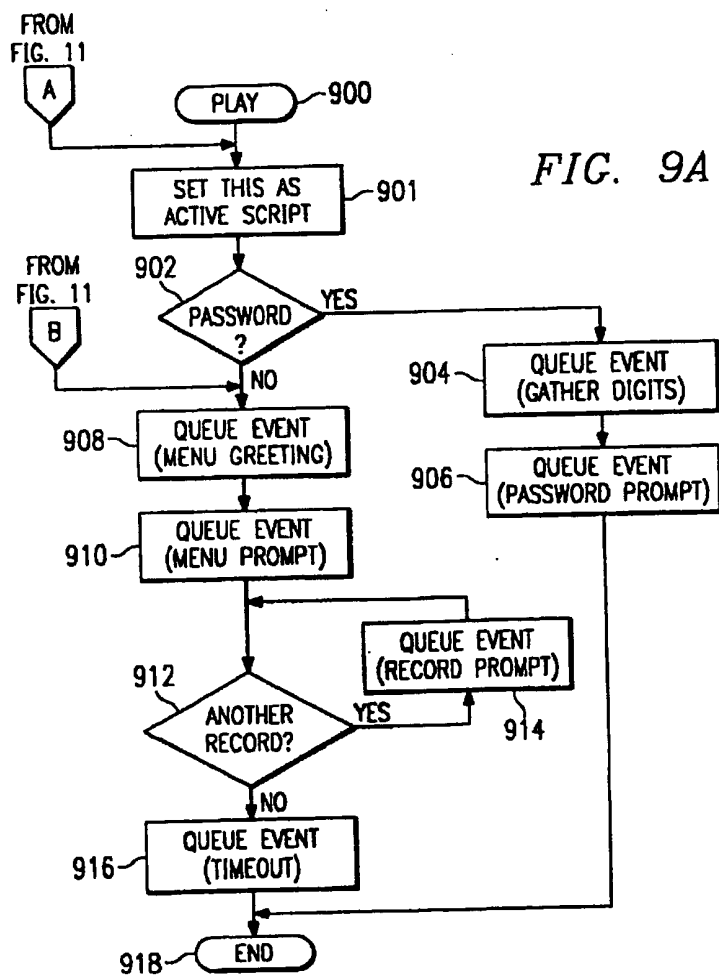


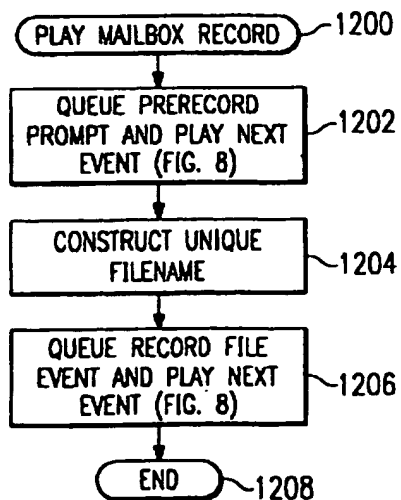
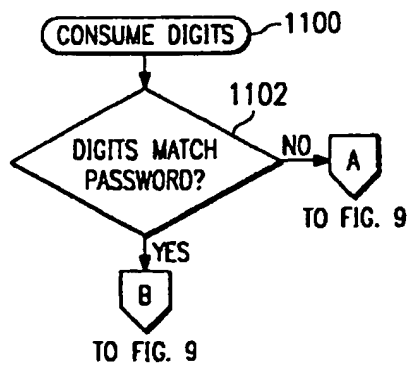
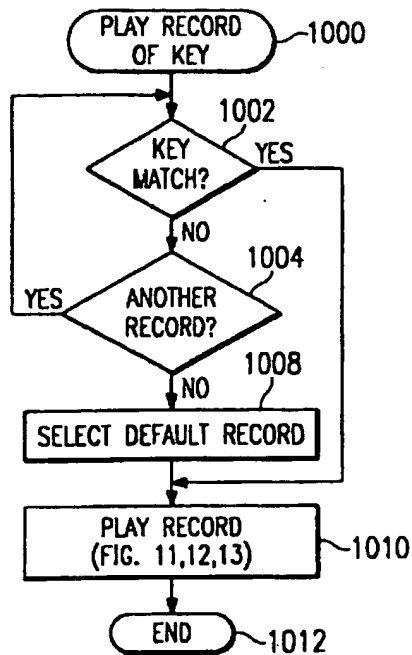
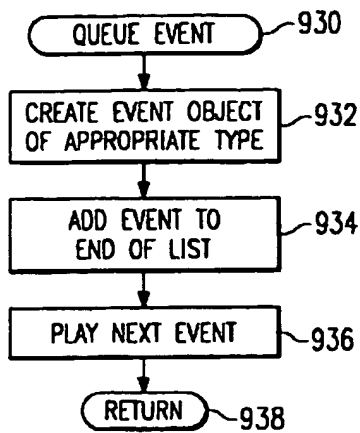
FIG. 9A

U.S. Patent

Apr. 7, 1998

Sheet 7 of 8

5,737,393



U.S. Patent

Apr. 7, 1998

Sheet 8 of 8

5,737,393

